Science of Computer Programming 118 (2016) 3–23

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Formal specification and verification of TCP extended with the Window Scale Option

Lars Lockefeer, David M. Williams, Wan Fokkink *

Department of Computer Science, VU University, Faculty of Sciences, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Received 6 January 2015
Received in revised form 19 July 2015
Accepted 10 August 2015
Available online 12 August 2015

Keywords:
μCRL
Process algebra
Transmission control protocol
Window scale option
Sliding window protocol

ABSTRACT

We formally verify that TCP satisfies its requirements when extended with the Window Scale Option. With the aid of our μCRL specification and the `LTSmin` toolset, we verify that our specification of unidirectional TCP Data Transfer extended with the Window Scale Option does not deadlock, and that its external behaviour is branching bisimilar to a FIFO queue for a significantly large instance. Separately, we verify that a connection may only be closed if both entities accept the CLOSE call from the Application Layer. Finally, we recommend a rewording of the specification regarding how a zero window is probed, ensuring deadlocks do not arise as a result of misinterpretation.

1. Introduction

The Transmission Control Protocol (TCP) plays an important role in the Internet, providing reliable transport of messages through a possibly faulty medium to many of its applications. Our primary contribution is the formal verification of TCP extended with the Window Scale Option; an option not considered in earlier verification efforts. We take care to extract our formal specification directly from the original specifications of TCP and the Window Scale Option, i.e., RFCs 793 [1], 1122 [2] and 1323 [3]. This work was initially triggered by a concern of Dr. Barry M. Cook, CTO at 4Links Limited, regarding the Window Scale Option proposed in RFC 1323. Specifically, he questioned whether the window size being reportable only in units of 2^n bytes conflicts with the requirement that the receive buffer space available should not change downward.

We adopt the process algebra μCRL as our formal specification language. Based on ACP, μCRL is enriched with the algebraic specification of abstract data types. We found μCRL's treatment of data as a first class citizen essential for specifying TCP, and were encouraged by its previous success in verifying the Sliding Window Protocol [4,5]. We utilise the μCRL toolset and `LTSmin` [6] to explicitly generate the state space and perform the automated verification.

Section 2 relates our verification effort to those that precede it. In Section 3 we present the functional specification of TCP, followed by an introduction of μCRL in Section 4. Section 5 presents our μCRL specification of TCP; its structure mirrors that of the functional specification presented in Section 3 and illustrated in Figs. 2 and 3. We split our verification across Sections 6 and 7 and conclude that the Window Scale Option does not adversely impact TCP. However, in Section 6.2

* Corresponding author. Tel.: +31 20 598 7735.
E-mail addresses: info@larslockefeer.nl (L. Lockefeer), w.j.fokkink@vu.nl (W. Fokkink).

http://dx.doi.org/10.1016/j.scico.2015.08.005
0167-6423/© 2015 Elsevier B.V. All rights reserved.

Table 1
Comparison of earlier verifications of TCP.

Authors	RFC 793	RFC 1122	RFC 1323	Other extensions	Conn Establishment	Conn Teardown	Data Transfer	Message loss	Duplication	Reordering	Message direction	Conn incarnations	Window Scale
Murphy & Shankar [8]	√				√	√		√	√	√	⇔	*n*	
Smith [9,10], Smith & Ramakrishnan [11]	√			√	√	√	√	√	√	√	⇒	*n*	
Schieferdecker [12]	√	√		√	√	√				√	⇔	2	
Billington & Han [15–18]	√	√			√	√	√	√		√	⇔	1	
Bishop et al. [14], Ridge et al. [19]	√	√	√	√	√	√	√	√	√	√	⇔	*n*	
Our verification	√	√	√			√	√	√	√	√	⇒	1	√

we recommend a reformulation of RFC 793 to avoid deadlocks that can arise due to the ambiguous formulation of how to probe zero windows.

An earlier version of this paper appeared as [7], where only the architecture of our models was presented and our verification focused solely on Data Transfer. Here, we provide our μCRL specification in sufficient detail for the results of our verification to be reproduced. Moreover, in Section 7, next to the Window Scale Option we also include Connection Teardown to show that the connection can only be closed if both entities accept the CLOSE call from the Application Layer.

2. Related work

Table 1 compares our verification of TCP to previous efforts, which we will discuss in this section. Murphy & Shankar [8] specified a protocol with a similar service specification to TCP as defined in RFC 793. By a method of step-wise refinement, a protocol was specified maintaining several correctness properties. The need for a three-way handshake and strictly increasing incarnation numbers becomes apparent with the introduction of each fault in the medium. Similarly, by means of a refinement mapping, Smith [9,10] has shown that the protocol satisfies the specification of the user-visible behaviour. Selective acknowledgements were added in [11]. Schieferdecker [12] showed that there is an error in TCP's handling of the ABORT call. After proposing a solution, a LOTOS specification of TCP was given and several μ-calculus properties were verified using CADP [13]. Bishop et al. [14] considered whether execution traces generated from real-world implementations of TCP were accepted by a Higher Order Logic (HOL) specification of TCP including Protection Against Wrapped Sequence Numbers (PAWS), the Window Scale Option and congestion control algorithms. Of the test traces generated, the specification accepted 91.7%.

Billington & Han have studied TCP extensively considering both RFC 793 and RFC 1122 using Coloured Petri Nets. They have given a concise overview of their TCP service specification in [15] which includes connection establishment, normal data transfer, urgent data transfer, graceful connection release and abortion of connections. In [16], they gave a model of the connection management service, which was further refined in [17]. This revised specification was used as a basis for a verification of connection management [18] considering a model without retransmissions and a model with retransmissions. As a result of their verification efforts, Billington & Han found several issues within connection management. For example, in the model without retransmissions, a deadlock could occur when one entity opened the connection passively and, after receiving and acknowledging a connection request, immediately closed the connection again. This deadlock occurred even on a non-lossy network and could be resolved by introducing retransmissions. The authors noted that it is strange that retransmissions of messages are required in the case of a non-lossy medium. The work by Billington & Han on Data Transfer has not yet led to a verification.

As the Sliding Window Protocol (SWP) underlies TCP (see Section 5) we also compare our verification to those of SWP. We, like Bezem & Groote [20] and Badban et al. [4], use μCRL for our specification. Bezem & Groote and Badban et al. consider bidirectional communication across a medium that can lose but neither duplicate nor reorder messages. Our verification considers a medium that can lose, duplicate and reorder messages, but does so only in a unidirectional setting. Whereas we, like Bezem & Groote, consider a finite window size (namely 2^2), Badban et al. performed the verification on an arbitrarily large window. Madelaine & Vergamini [21] modelled and verified SWP using LOTOS and AUTO. They, like us, consider the unidirectional case across a medium that can lose, duplicate and reorder messages. Finally, Chkliaev et al. [22] specify an amended version of SWP, in which the sender and receiver need not synchronise on the sequence number initially.

3. Functional specification of TCP

The Transmission Control Protocol (TCP) enables two parties to reliably communicate over a faulty network. Its responsibilities can roughly be divided into two categories: Connection management sets up the connections, manages the connection states and ensures that connections are closed in a safe manner; Data transmission involves the transfer of segments from the sender to the receiver. In this section we present TCP as specified in RFCs 793 and 1122, extended to include the Window Scale Option of RFC 1323. We focus on the Data Transfer and Connection Teardown phases, as it is these that we formally specify in μCRL in Section 5 and verify in Sections 6 and 7.

The TCP instance of the sender receives data from some application and packages this into segments to be handed to the Network Layer. The TCP instance of the receiver receives segments from the Network Layer and should ensure the data is delivered to the receiver's application in the same order as it was sent. To prevent segments from lingering around the network forever, a Maximum Segment Lifetime is defined. Every time a segment arrives at a hop in the network, the hop verifies whether the time that has expired since the message was sent is smaller than its maximum lifetime. If this is not the case, the segment is removed from the network by the hop. TCP may send the octets in its buffer at its own convenience. As this behaviour may lead to undesirable delays, two mechanisms are available for the Application Layer to indicate that the data that it wants to send is important: the PUSH function indicating that the sender must transmit the data immediately and the URGENT function, indicating that the data must be processed by the receiver as soon as possible upon arrival.

The purpose of a segment is twofold: (i) a segment may contain zero or more octets of data that the sender's application wishes to relay to an application at the receiver; and, (ii) a segment communicates control information between the two entities. This control information consists of several variables, flags and options.

Control variables: SEG.SRC and SEG.DST specify the port numbers that the TCP sender and receiver use. If the SYN-flag or FIN-flag is not set, the SEG.SEQ field contains the sequence number of the first octet. SEG.ACK specifies the next sequence number that the sender expects to receive. This field is only to be interpreted if the ACK-flag is set. The data offset field, SEG.OFF, specifies the size of the TCP header as a multiple of 32, indicating where the data begins. SEG.WND specifies the number of octets that the sender is willing to accept. SEG.CHK specifies a checksum calculated over the header and data by the sender to facilitate integrity checking by the receiver. Finally SEG.UP contains the sequence number of the last octet that is marked as urgent. This field is only interpreted if the URG-flag is set.

Control flags: SEG.URG indicates that the urgent function is triggered at the sender of the segment; SEG.ACK indicates that the segment contains acknowledgement information; SEG.PSH indicates that the push function is triggered at the sender of the segment; SEG.RST indicates that the reset function is triggered at the sender of the segment; SEG.SYN indicates that both entities are synchronising on an initial sequence number; SEG.FIN, indicates that no more data will come from the sender and that it wishes to close the connection.

Options: To facilitate enhancements to TCP without breaking the core specification, options may be appended to the end of the header. One such option is the Window Scale Option.

TCP uses the Sliding Window Protocol (SWP) for its data transfer. Both sender and receiver maintain a window of n sequence numbers, ranging from 0 to $n - 1$, from which they can send or where they can receive data items. The sender may send as many octets as the size of its window before it has to wait for an acknowledgement from the receiver. Once the receiver sends an acknowledgement for m octets, its window *slides* forward m sequence numbers. Likewise, the sender's window *slides* m sequence numbers if this acknowledgement arrives. To function correctly over mediums that may lose data, the maximum size of the window is $\frac{n}{2}$ [23].

In the implementation of SWP underlying TCP, octets may be acknowledged before they are forwarded to the Application Layer (AL) and therefore still occupy a position in the receive buffer. In this case, the receiving entity reduces its advertised window through the SEG.WND field of the acknowledgement segment, ensuring the sending entity does not send new data that will overflow its buffer. Once the octets are forwarded to the Application Layer, it may reopen the window. The receiver may adjust the size of the sender's window at any time, through the value of SEG.WND set in acknowledgement segments. As the size of this field is limited to 16 bits, TCP can send at most 2^{16} octets into the medium before having to wait for an acknowledgement, and if the medium can hold more octets an unnecessary delay will be incurred. To resolve this, RFC 1323 [3] proposes the Window Scale Option. This option will be discussed in detail in Section 3.3.2; the Window Scale Option is of primary concern in our verification of TCP.

3.1. Connection management

TCP begins by establishing a connection with both entities reaching agreement on the configuration to use for the connection that is stored in their Transmission Control Block (TCB). In TCP, each connection is bidirectional, meaning that each TCP entity may act as both sender and receiver. Regarding outgoing data, the TCB maintains the following variables: A pointer to the send buffer, which contains octets accepted as a result of a SEND call from the Application Layer; the retransmission queue, which points to a queue holding sent segments until they are acknowledged; SND.UNA, the sequence number of the first octet sent but not yet acknowledged; SND.NXT, the sequence number of the first octet next to be sent; SND.WND, the total number of octets transmission allows; SND.UP, the sequence number of the first octet following the data marked as urgent; SND.WL1, the sequence number of the segment used for the last window update; SND.WL2, the acknowledgement number of the segment used for the last window update; ISS, the initial send sequence number, i.e., the sequence number of the first segment the entity will send. In addition, the following variables are maintained regarding incoming data: a pointer to the receive buffer, a buffer in which octets accepted from the Network Layer are stored before

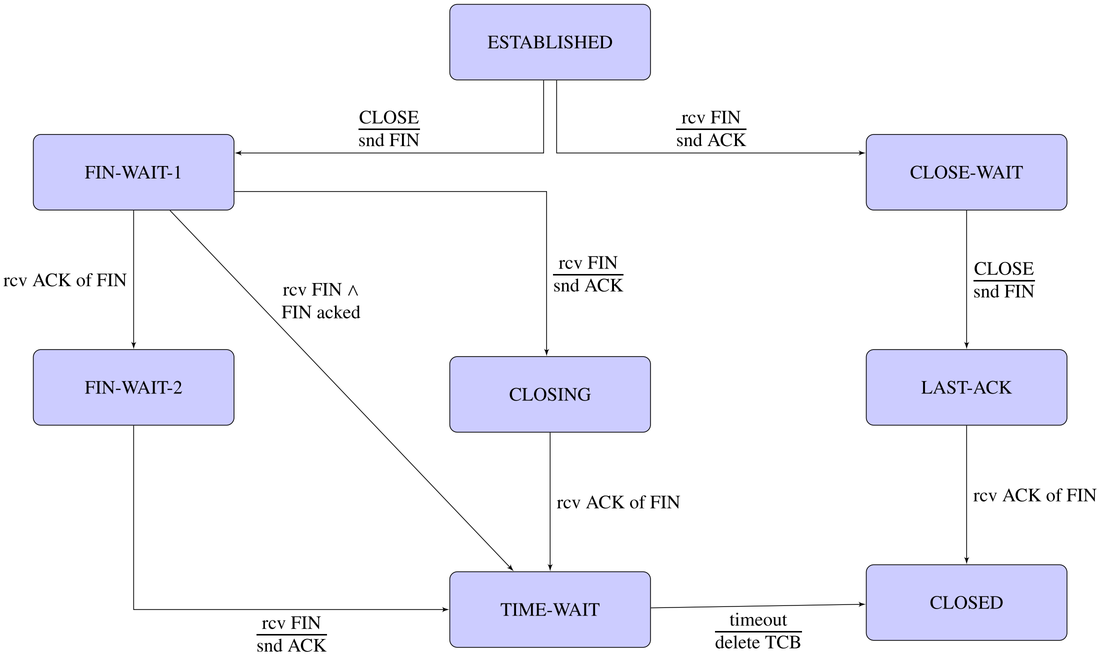

Fig. 1. The Connection Teardown procedure.

being forwarded to the Application Layer; RCV.NXT, the sequence number of the next segment the receiver expects to receive; RCV.WND, the maximum number of octets the entity is prepared to accept at once; RCV.UP, the sequence number of the first octet following the data marked as urgent; IRS, the initial receive sequence number, i.e., the sequence number of the first expected segment.

During its lifetime, a connection progresses through several states. Of course, both initially and after finishing the communication the connection does not exist. In this case, the connection's state is described as CLOSED. Connection establishment is initiated by issuing the OPEN call from the Application Layer to TCP. Once a connection has been set up between two entities, the state will be set to ESTABLISHED.

Unfortunately, nothing lasts forever and therefore, a Connection Teardown mechanism is also specified. During this procedure, the protocol progresses through several states of which an overview is given in Fig. 1. As soon as an application at an entity has no more data to send, it issues the CLOSE call from the Application Layer. It is important to note that as a result of issuing this call, the data transfer flowing from the TCP entity that issued the CLOSE call to the remote TCP entity will be terminated, essentially transforming the bidirectional connection into a unidirectional one. Only after a CLOSE call has been issued from the Application Layer of the remote TCP entity as well, the connection will be torn down completely. Hence, there are two scenarios that need to be discussed:

***While in the ESTABLISHED state, the local TCP entity receives a CLOSE call from the Application Layer**:* Upon receiving a CLOSE call from the Application Layer, the TCP entity will delay the processing of this call until any byte it has buffered in its send buffer is segmented and sent to the entity at the other end. Then, it will send a segment with the FIN flag set, after which the connection will progress from the ESTABLISHED state to the FINWAIT-1 state. While in this state, the TCP entity will no longer accept any SEND calls from the Application Layer, and wait for an acknowledgement of the FIN segment to arrive. Once an acknowledgement of the FIN segment is received, the connection will progress to the FINWAIT-2 state. The connection will remain in the FINWAIT-2 state until the TCP entity receives a FIN segment from the other end, indicating that at the other end of the connection, a CLOSE call was issued from the Application Layer. At this point, the TCP entity that received the FIN will send an acknowledgement and progress to the TIME WAIT state. This state adds a delay of twice the Maximum Segment Lifetime (MSL) before the connection is definitely closed, to ensure that the acknowledgement arrives at the other side. Finally, the connection progresses to the CLOSED state, meaning that all state information for the connection is deleted from the TCP entity. There is a slight variation to this scenario where the TCP entity receives a segment with the FIN flag set while in the FIN WAIT-1 state, indicating that at the remote entity, a CLOSE call was issued from the Application Layer as well. In this case, the TCP entity sends an acknowledgement and progresses its state to CLOSING. Upon receiving an acknowledgement of its own FIN segment, the entity will progress to the TIME WAIT state.

***While in the ESTABLISHED state, the local TCP entity receives a FIN from the network**:* Upon receiving a segment with the FIN flag set, the TCP entity will send an acknowledgement and progress to the CLOSE-WAIT state. In this state, the TCP entity will no longer accept RECEIVE calls from the Application Layer but may still accept SEND calls. The TCP entity remains

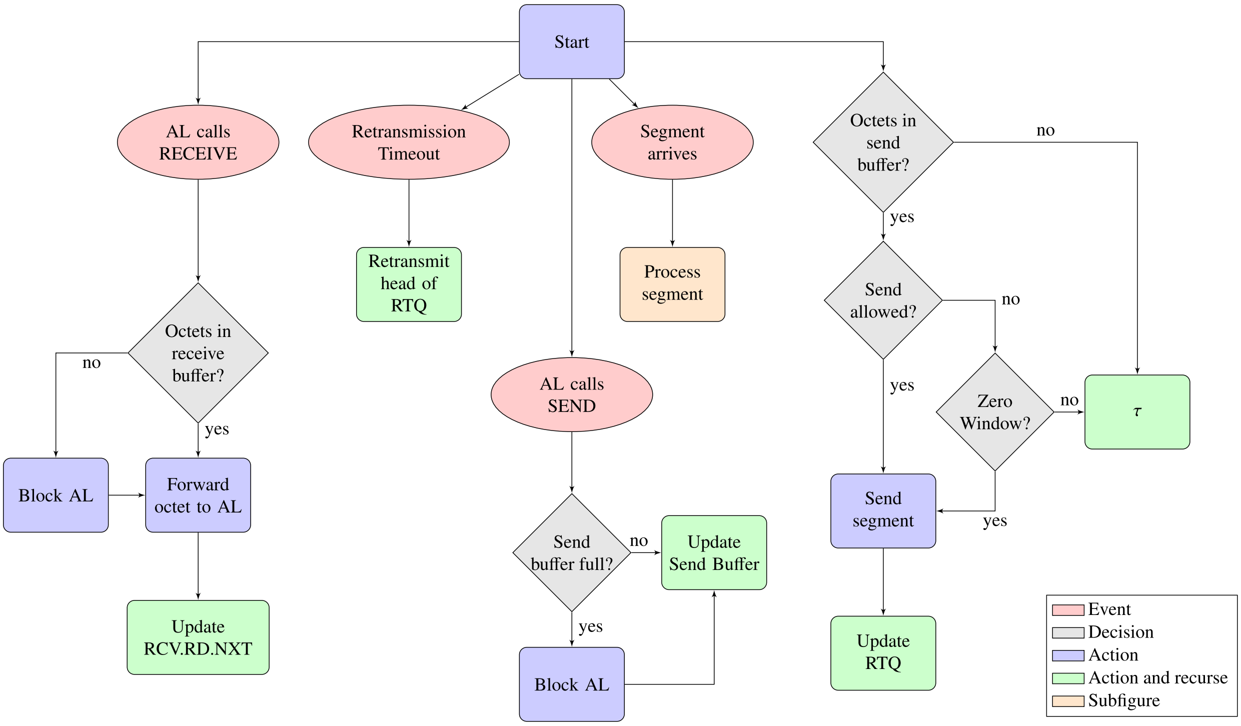

Fig. 2. Abstract overview of our specification of TCP Data Transfer.

in this state until a CLOSE call is issued from the Application Layer. Upon receiving this call, the TCP entity will send a segment with the FIN flag set and progress to the LAST-ACK state, waiting for an acknowledgement of the segment it just sent. Once this acknowledgement is received, the connection progresses to the CLOSED state.

3.2. Data Transfer

Between the Connection Establishment and Connection Teardown phase, data can be transferred between TCP entities. To simplify our discussion, we will distinguish between a sender and a receiver that engage in a unidirectional transfer of data. In a real-world scenario, data would flow in both directions, requiring the sender to also act as a receiver and vice versa. The structure of the remainder of this section reflects the high-level overview of TCP, as illustrated in Fig. 2. Fig. 3 illustrates the *Process segment* procedure in finer granularity. In both figures, Application Layer is abbreviated as AL.

3.2.1. Application Layer calls send

Data transfer starts at the Application Layer of the sender, where octets of data that are to be sent to the remote entity can be passed to TCP by (consecutive) SEND calls, which may be issued as long as the connection is in the ESTABLISHED or CLOSE-WAIT state. The sender maintains a buffer of these octets, the send buffer, which operates as a FIFO queue. As long as there is capacity left in the send buffer, TCP will accept SEND calls from the Application Layer and put the octets that are passed as arguments to these SEND calls in the buffer. TCP may send the octets in its buffer at its own convenience. After a single SEND call, it could for example wait for more SEND calls from the Application Layer before sending out any data.

Some ambiguity surrounds the specification of the sequence number, as both octets and segments are assigned one. In principle, TCP numbers each octet with a unique sequence number, modulo the size of the sequence number space. A segment inherits its sequence number from the first octet it contains. However, a segment containing no octets still requires a sequence number. Here, it is still numbered with the sequence number maintained in SND.NXT, but SND.NXT is not updated.

SYN and FIN segments, which are used during connection setup and teardown, form an exception to this rule, as is stated on page 26 of [1]: *“The SYN segment is considered to occur before the first actual data octet of the segment in which it occurs [if any], while the FIN is considered to occur after the last actual data octet in a segment in which it occurs.”* Hence, if a SYN segment is sent with sequence number n, this same sequence number must not be used to send a data segment after this SYN segment until the sequence number space wraps. Likewise, if the last byte that is sent on a connection has sequence number n, the FIN segment that is subsequently used to close the connection gets sequence number $n + 1$. In both cases, SND.NXT is updated accordingly after sending the control segment.

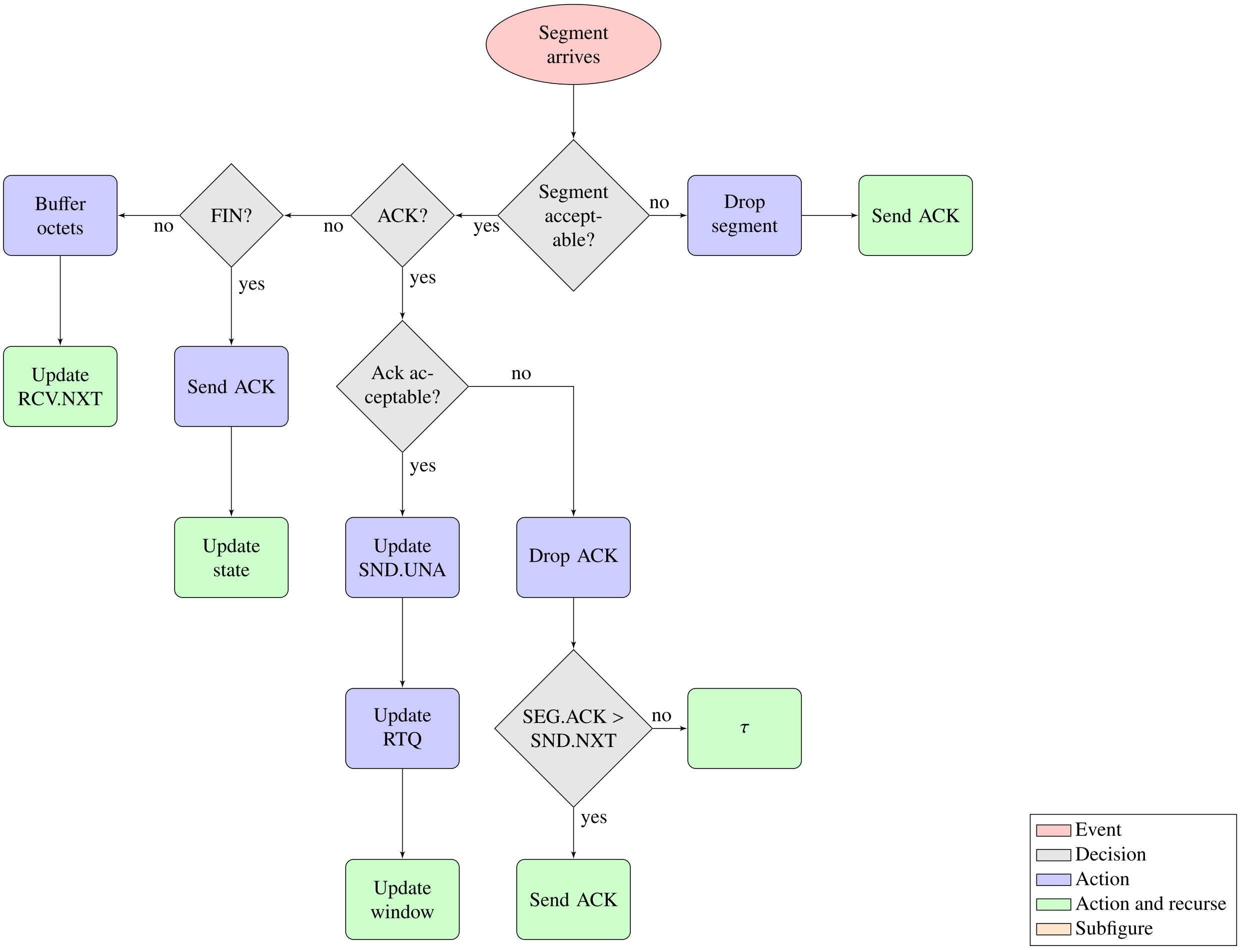

Fig. 3. Abstract overview of our specification of TCP Segment Processing.

3.2.2. Octets in send buffer?

Once there are octets in the send buffer, TCP may package them up into a segment. The actual number of octets that TCP can send at a certain point in time is calculated by taking the difference m between SND.UNA and SND.NXT. If $m <$ SND.WND, TCP may package any number of octets $n \leq m$ that it thinks reasonable into a segment and send it into the medium. Subsequently, TCP does several things:

1. The octets that were included in the segment are removed from the send buffer.
2. The segment that was sent is put on the retransmission queue.
3. A retransmission timer is started for the segment.
4. SND.NXT is advanced by n, now indicating the sequence number of the octet that will be sent next.

3.2.3. Segment arrives

If all goes well, after the transfer through the medium a segment will eventually arrive at the receiver. Recall that in its TCB, the receiver maintains a pointer to the receive buffer and several variables. Of importance here are the variables RCV.NXT and RCV.WND. Initially, RCV.NXT is set to the initial sequence number that the sender has communicated to the receiver during connection setup and RCV.WND is set to the capacity of the receive buffer. As a first check on the segment that arrived, the receiver will verify that the segment is acceptable. A segment is deemed acceptable in the following two situations [1]:

1. If the segment does not contain data octets:
   (a) If $\texttt{RCV.WND} = 0$, it is required that $\texttt{SEG.SEQ} = \texttt{RCV.NXT}$
   (b) If $\texttt{RCV.WND} > 0$, it is required that $\texttt{RCV.NXT} \leq \texttt{SEG.SEQ} < \texttt{RCV.NXT} + \texttt{RCV.WND}$
2. If the segment does contain data octets

(a) If $\texttt{RCV.WND} = 0$, the segment is not acceptable.
(b) If $\texttt{RCV.WND} > 0$, it is required that
- either $\texttt{RCV.NXT} \leq \texttt{SEG.SEQ} < \texttt{RCV.NXT} + \texttt{RCV.WND}$
- or: $\texttt{RCV.NXT} \leq \texttt{SEG.SEQ} + \texttt{SEG.LEN} -1 < \texttt{RCV.NXT} + \texttt{RCV.WND}$

When processing the segment, four things are of importance: the arrival of an unacceptable segment, whether the segment has the FIN flag set, whether it contains acknowledgement information and whether it is carrying data.

The segment is not acceptable If the segment is not acceptable, the segment is dropped and an acknowledgement is sent to the sender containing the current value of RCV.NXT. After the initial acceptability check, segments are processed in order of their sequence numbers. Segments that arrive out of order may be dropped by the receiver. However, to improve performance, the specification suggests that these segments are held in a special buffer to be processed as soon as their turn arrives.

If the segment is acceptable, the receiving instance will check whether any of the following control flags is set: RST, SYN, ACK and URG. Of these flags, the RST and SYN flags can only be set during connection setup or as a consequence of errors during connection setup. Therefore, we will not discuss the behaviour of the protocol in response to such a flag being set here.

The segment contains acknowledgement information If an acknowledgement arrives at the sender, the sender verifies whether $\texttt{SND.UNA} < \texttt{SEG.ACK} \leq \texttt{SND.NXT}$. If this is the case, SND.UNA is set to SEG.ACK and all segments on the retransmission queue that contain octets with sequence numbers $n \ldots m < \texttt{SEG.ACK}$ are removed. Furthermore, if $\texttt{SND.UNA} \leq \texttt{SEG.ACK} \leq \texttt{SND.NXT} \wedge (\texttt{SND.WL1} < \texttt{SEG.SEQ} \vee (\texttt{SND.WL1} = \texttt{SEG.SEQ} \wedge \texttt{SND.WL2} \leq \texttt{SEG.ACK}))$, the send window must be updated by setting SND.WND to SEG.WND, SND.WL1 to SEG.SEQ and SND.WL2 to SEG.ACK. If it is not the case that $\texttt{SND.UNA} < \texttt{SEG.ACK} \leq \texttt{SND.NXT}$, there are two possibilities. When $\texttt{SEG.ACK} \leq \texttt{SND.UNA}$, the acknowledgement can be ignored since it is a duplicate. When $\texttt{SEG.ACK} > \texttt{SND.NXT}$, the sender will send an acknowledgement, drop the segment and return. This last situation can only occur when data transfer is bidirectional.

The segment is carrying data If the segment is carrying data, the octets are then taken from the segment and written into a buffer at the receiver and RCV.NXT is advanced by the number of octets that have been accepted. The receiver must acknowledge the fact that it took responsibility for the data in the segment to the sender, and to this end, an acknowledgement containing the new value of RCV.NXT – reflecting the next sequence number that the receiver expects to receive – is constructed and sent back to the receiver. This sets the TCP implementation of SWP apart from other implementations, as an acknowledgement is sent while the octets may not yet be forwarded to the Application Layer and therefore occupy a position in the receive buffer. Therefore, the size of the window that is reported back to the sender represents the available capacity in the receive buffer, if this capacity is less than the difference between RCV.NXT and the size of the receive window that was originally agreed upon permits. In naive implementations, each acknowledgement will carry the updated size of the receiver's window. Several strategies have been proposed to minimise the performance degradation this causes, which is outside the scope of our work. Our verification shall concern the correctness of TCP and not its performance.

The segment has the FIN flag set Finally, the receiver will verify whether the FIN flag was set in the incoming segment. If this was the case the receiving instance will progress from the ESTABLISHED to the CLOSE-WAIT state once processing the segment has completed.

3.2.4. Retransmission timeout

If the retransmission timer expires before the sender receives an acknowledgement of the segment, the segment will be retransmitted and the timer restarted. By doing this, any segment that is not accepted at the remote end, for whatever reason, is retransmitted until it is eventually accepted exactly once. This process may repeat itself as long as $m < \texttt{SND.WND}$. By default, TCP uses a go-back-n retransmission scheme. However, the protocol may keep segments that arrive out of order to employ a selective repeat retransmission scheme. This scheme can be optimised even further by implementing the selective acknowledgement extension [24].

3.2.5. Probe zero window

In each acknowledgement, a receiving TCP instance may adjust the size of the send window of the remote entity. If the sender has a send window of size 0, this may lead to a deadlock, since the sender is not allowed to send anything and therefore will not receive any additional acknowledgements that may contain an updated window size. Therefore, a TCP instance must regularly transmit something to the remote entity to ensure that it will receive an acknowledgement with a possibly reopened window. This behaviour is called probing the zero window.

3.2.6. Application Layer calls RECEIVE

To complete the Data Transfer phase, the application at the receiving end must read the octets from the receive buffer of TCP. This is done by issuing the RECEIVE call, which may be issued as long as the connection is in the ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2 or CLOSE-WAIT state. Once an octet is forwarded to the Application Layer, its transfer is complete.

3.3. Known problems

Several issues with TCP's implementation have surfaced as the networks that the protocol operates over have become more sophisticated. In this section, we will discuss the problems that are relevant to our project.

3.3.1. Sequence number reuse

Since the sequence number field in the TCP header only allows for a finite 32-bit sequence number, the protocol can only use sequence numbers up until $2^{32} - 1$ to number the octets that it sends. As a result of this, after 2^{32} octets have been sent the next octet will again have a sequence number equal to the initial sequence number. In RFC 793, it is stated that *"the duplicate detection and sequencing algorithm in the TCP protocol relies on the unique binding of segment data to sequence space to the extent that sequence numbers will not cycle through all* 2^{32} *values before the segment data bound to those sequence numbers has been delivered and acknowledged by the receiver and all duplicate copies of the segment have 'drained' from the internet"* [1]. Such 'draining' is achieved by enforcing the Maximum Segment Lifetime (MSL) at each hop in the network. Since TCP has a sequence number space of size 2^{32}, this assumption is automatically met for a reasonable MSL of two minutes and networks up to a speed of 17.9 megabytes per second. However, as network speeds increased, scenarios arose where a sender could send its entire sequence number space into the network in an amount of time shorter than the MSL. Protection Against Wrapped Sequence Numbers (PAWS) is proposed in RFC 1323 [3] to resolve this problem.

If PAWS is implemented, to every segment a timestamp `SEG.TSval` is added that is monotone non-decreasing in time. Furthermore, the receiver maintains the additional variables `TSrecent` and `Last.ACK.sent` in its TCB. Whenever an acknowledgement segment is sent, `Last.ACK.sent` is set to the value of `SEG.ACK`. If a segment arrives at the receiver for which $\texttt{SEG.TSval} < \texttt{TSrecent}$, the segment is dropped and an acknowledgement is sent to the sender containing the current value of `RCV.NXT`. If $\texttt{SEG.TSval} \geq \texttt{TSrecent}$ and $\texttt{SEG.SEQ} \leq \texttt{Last.ACK.sent}$, `SEG.TSval` is stored in `TSrecent`. Regardless of the outcome of this test, processing continues as specified in RFC 793 [1].

By implementing PAWS the sequence number of a segment is transformed from a single value into a two-tuple. It is important to note here that these timestamps are themselves 32-bit unsigned integers in a modular 32-bit space (again due to the restrictions of the TCP header). Hence, the problem is only moved forward. There is no other protection against wrapped sequence numbers than the assumption that whenever a connection enters a fragment of the sequence number space for the $(n+1)$th time, all segments that were sent into the network while the connection was in the same fragment of the sequence number space for the nth time have drained from the network due to the expiry of their MSL. By choosing the values for the timestamp clock wisely, the implementation can be stretched to cover any value for the MSL that is still reasonable.

3.3.2. Performance loss due to small window size

There is a direct relation between TCP's performance and the size of the send window: the larger this window is, the more data a sender can send without having to wait for an acknowledgement. In an optimal scenario, the size of the window allows the sender to send as much data into the medium as it can hold at most. However, the receiver may adjust the size of the sender's window at any time, through the value of `SEG.WND` set in acknowledgement segments that are transferred from receiver to sender. By the fact that the size of this field is limited to 16 bits, the maximum size of the send window is 2^{16}. This means that TCP can send at most 2^{16} octets into the medium before having to wait for an acknowledgement. Hence, if the medium can hold more than 2^{16} octets, unnecessary delay will be introduced into the communication due to the restriction on the window size.

To resolve this issue, RFC 1323 proposes the Window Scale Option. If implemented, the send and receive windows are maintained as 32-bit numbers in the TCB of the sender and receiver, which is also extended to include variables `SND.WND.SCALE` and `RCV.WND.SCALE`. Whenever an entity receives an acknowledgement, it left-shifts the value of `SEG.WND` by the value of `SND.WND.SCALE` before it updates its send window. Likewise, whenever an entity sends an acknowledgement it sets the window field of the outgoing segment to the size of its receive window, right-shifted by the scale factor `RCV.WND.SCALE`.

Implementing the Window Scale Option theoretically enables a window size that is equal to the size of the sequence number space. Therefore, it introduces an additional problem with sequence number reuse that did not occur previously. According to [23], the sliding window protocol functions correctly for window sizes up until 2^{n-1} given a sequence number space of size 2^n, assuming that the medium does not support reordering. However, in the environment in which TCP is used, this assumption does not hold. In [22], a scenario is given where a segment s_0 ranging over the first half of the sequence number space is erroneously accepted twice as a result of duplicating s_0 and a subsequent reordering with a segment s_1 ranging over the second half of the sequence number space.

Another example of the problem that may occur as a result of the fact that the assumption as given in [23] does not hold in TCP's context is shown by the following scenario, in which we have a sequence number space of size 2^3 and a window of size 2^2. The sender starts by sending a segment x containing octets $0 \ldots 3$. After receiving an acknowledgement of this segment, the sender responds by sending a segment x' containing octets $4 \ldots 7$. Again, this segment is acknowledged, after which the sender will send a segment y containing octets $0 \ldots 3$. At this point, the receiver's window ranges over octets $0 \ldots 3$. If segment y arrives, the receiver will accept it, update the window to range over octets $4 \ldots 7$ and send an acknowledgement. Before this acknowledgement arrives at the sender, segment y is retransmitted. Immediately thereafter, the sender receives the acknowledgement, updates its window and sends a segment z containing octets $4 \ldots 7$. Now, let segment z overtake the retransmitted segment y in the medium and arrive at the receiver. The receiver will accept the segment, update its window range to $0 \ldots 3$ and send an acknowledgement. Shortly thereafter, the retransmitted segment y arrives. This segment is now accepted as a regular, in-sequence segment resulting in a corrupted byte stream.

Enforcing the assumption on the MSL does not help us here since segment z is sent shortly after segment y was retransmitted. Therefore, this scenario is completely reasonable. To fix this issue, RFC 1323 enforces that the window size is at most 2^{n-2} with n the number of bits available for the sequence number. Now, when the sender retransmits a segment y carrying octets $6 \ldots 7$ and shortly thereafter receives an acknowledgement for this segment, it will respond by sending a segment z carrying octets $0 \ldots 1$. If z overtakes y and arrives at the receiver, its window will be updated to range over octets $2 \ldots 3$. As a result of this, segment y will not be accepted if it were to arrive. Combined with the assumption that segment y will have drained from the network by the time that the receiver's window ranges over $0 \ldots 1$ again, correctness is preserved.

4. μCRL

Process algebras are used to formally specify the behaviour of (concurrent) systems. In general, first the separate components that make up the system are specified. Then, the components are put in parallel, together with a specification of ways for the components to interact with each other. Finally, the initial state of the parallel specification is denoted. The formal specification of the system's behaviour can then be used to verify the correctness of the system. We use both process equivalence, where (a part of) the behaviour is compared with the behaviour of another system, and property checking, where properties of the system are checked on its state space. μCRL distinguishes itself from other process algebras through its ability to cope with general abstract data types. For an overview of the most prominent formal verification techniques for communication protocols using μCRL, the reader is referred to [25]. For a concise overview of formal methods and their applications in software verification, we refer to [26].

4.1. Process terms

The most atomic form of a process term is an *action*. An action may carry zero or more data parameters, indicating the data that is relevant for the execution of the action. Process terms may be composed to form more intricate behaviour. A *sequential composition* of two process terms t_1 and t_2, denoted $t_1 \cdot t_2$, represents the process that first executes the process as described by term t_1 and, after successful termination, executes the process as described by term t_2. An *alternative composition* of two process terms t_1 and t_2, denoted $t_1 + t_2$, represents the process that executes either the process as described by term t_1 or the process as described by the term t_2. A process term can also reflect a deterministic choice based on a condition through composition with the *conditional operator*. A process term of the form $p \lhd b \rhd q$ with p and q process terms and b a boolean condition behaves a p if b evaluates to true and as q otherwise. We adopt the convention that the $\cdot$ operator binds stronger than the $+$ operator. The conditional operator binds stronger than $+$ and weaker than $\cdot$.

When a system is made up of multiple components, these components will in general work alongside each other and communicate from time to time to influence the behaviour of the other components. To this end, process algebras also allow process terms to be put in parallel. The first parallel operator is *merge*, denoted $\|$, that represents two process terms working alongside each other. If two process terms $p = a$ and $q = b$, consisting of the execution of action a or b respectively, are merged and no communication is possible between the two terms, the resulting process term will behave as the arbitrary interleaving of their actions. Hence, $p \| q$ behaves as $a \cdot b + b \cdot a$.

Certain actions in processes p and q may be synchronised using the *communication* operator $|$. If $p = a$ and $q = b$ are merged as before, we may additionally specify that the actions a and b synchronise: $a|b = c$. The parallel composition of the process terms $p \| q$ behave as $a \cdot b + b \cdot a + c$ (where $\cdot$ binds stronger than $+$). To enforce that the actions may only occur synchronously, actions may be *encapsulated*. By encapsulating a and b in $p \| q$ only the action c can occur. Encapsulation requires an additional action δ called deadlock. δ does not display any behaviour and is specified such that $p + \delta = p$ and $\delta \cdot p = \delta$. Encapsulation now works by substituting the atomic actions that make up a communication action with δ such that only the synchronous behaviour is exposed.

4.2. Process declarations

Process declarations are always of the form $P(x_1 : D_1, \ldots, x_n : D_n) = p$ with $n \geq 0$. This declares the process P that takes data variables $x_1 \ldots x_n$ as parameters and behaves as the process term p. p may contain occurrences $Q(y_1, \ldots, y_n)$ that further specify the process to be executed. It may also contain a recursive call to P, as long as the recursive call is *guarded*, meaning that it is preceded by an action. By using the *sum operator* $\sum_{d_1:D_1,\ldots,d_n:D_n} P(d_1, \ldots, d_n)$, a process term $P(d_1, \ldots, d_n)$ can be specified for any permutation of datum parameters $d_1 : D_1, \ldots, d_n : D_n$. In the resulting state space, this operator is reflected by a parallel composition over all possible parameterisations.

Actions may be hidden through the use of the *hiding operator* τ_A. If this operator is applied to a process term p, all actions $a \in A$ will be substituted with the special action τ. This special action name is used for actions that are not observable or not of interest for the specification. Sometimes, however, the presence of τ actions can be observed as a result of the composition of process terms. In the process term $a + \tau \cdot b$, for example, the τ action is of interest; once it is executed, the set of possible behaviours of the system is reduced from $\{a + b, b\}$ to $\{b\}$. Such a τ-action is called *non-inert*. Conversely, an *inert* τ-action is an action that does not lose any possible behaviours.

4.3. Process equivalence

Correctness of a system can be verified by checking whether a Labelled Transition System (LTS) generated from a process declaration is equivalent to a process capturing its requirements. Many notions of process equivalence have been proposed, a hierarchy of these is presented in [27,28]. *Bisimulation*, originally defined in [29], is located at the finest end of the spectrum. In [27] it is defined as: a binary relation R on processes, such that, for $a \in Act$

1. if pRq and $p \xrightarrow{a} p'$, then $\exists q' : q \xrightarrow{a} q'$ and $p'Rq'$
2. if pRq and $q \xrightarrow{a} q'$, then $\exists p' : p \xrightarrow{a} p'$ and $p'Rq'$

Here, p, p', q, q' denote processes and Act the set of possible actions. Two processes p and q are said to be bisimilar, $p \leftrightarroweq q$, if there exists a bisimulation relation R such that pRq.

A refinement of bisimulation is *branching bisimulation* [30], which takes τ-transitions into account in the equivalence relation. Intuitively, inert τ transitions do not have to be performed by process p as well as q for p and q to be branching bisimilar $p \leftrightarroweq_B q$. A *branching bisimulation relation* is: a binary relation R on processes, such that:

1. if pRq and $p \xrightarrow{a} p'$, then
   (a) either $a = \tau$ and $p'Rq$
   (b) or $\exists q'' : q \xrightarrow{\tau} \ldots \xrightarrow{\tau} q''$ for zero or more τ transitions, such that pRq' and $q'' \xrightarrow{a} q'$ with $p'Rq'$
2. if pRq and $q \xrightarrow{a} q'$, then
   (a) either $a = \tau$ and pRq'
   (b) or $\exists p'' : p \xrightarrow{\tau} \ldots \xrightarrow{\tau} p''$ for zero or more τ transitions, such that $p''Rq$ and $p'' \xrightarrow{a} p'$ with $p'Rq'$

Here p, p', p'', q, q', q'' denote processes and Act the set of possible actions. Two processes p and q are said to be branching bisimilar, $p \leftrightarroweq_B q$, if there exists a branching bisimulation relation R such that pRq. Branching bisimulation equivalence implicitly enforces a notion of *fairness* on processes when comparing them. Intuitively, this notion ensures that if an exit transition exists from a τ-loop, then this transition will eventually be taken. Several fairness notions exist, for an overview we refer to [31].

As the complexity of processes increases, the size of the state space tends to grow exponentially. Minimisation techniques have been proposed to resolve this issue. One of these techniques is minimisation modulo branching bisimilarity, that prunes inert τ-transitions from a state space, for which an efficient algorithm was proposed in [32]. As τ-transitions are pruned from the state space, a fairness assumption is again enforced on the state space. If $P \leftrightarroweq_B Q$ and both P and Q are minimised modulo branching bisimilarity yielding processes P' and Q', it holds that $P' \leftrightarroweq Q'$.

4.4. Property checking

Another approach for verifying the correctness of a process is to formulate properties and subsequently check that these properties hold on the state space generated of the process. A distinction is made between *liveness* and *safety* properties. A *liveness* property states that something 'good' will *eventually* happen, whereas a *safety* property states that something 'bad' will *never* happen. Properties may be formulated in μ-calculus [33], which is defined by the following BNF grammar:

$$\phi ::= \mathtt{T} \mid \mathtt{F} \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \langle a \rangle \phi \mid [a]\phi \mid X \mid \mu X.\phi \mid \nu X.\phi$$

$\langle a \rangle \phi$ holds for a state s if there exists a state s' in which ϕ holds and $s \xrightarrow{a} s'$. $[a]\phi$ holds for a state s if for all transitions $s \xrightarrow{a} s'$, ϕ holds in s'. $\mathtt{T}$ holds in all states while $\mathtt{F}$ does not hold in any state. The set X ranges over recursion variables. Minimal and maximal fixpoints $\mu X.\phi$ and $\nu X.\phi$ exist because μ-calculus formulas are monotonic. Here, ϕ represents a mapping that yields a set of states for which the property ϕ holds, ranging over the domain S of states in which the recursion variable X holds.

As an extension to the μ-calculus, the regular μ-calculus was proposed [34] in which instead of formulas $\langle a \rangle \phi$ and $[a]\phi$, one may use formulas $\langle \beta \rangle \phi$ and $[\beta]\phi$ with β a regular expression defined by the following BNF grammar:

$$\alpha ::= \mathtt{T} \mid a \mid \neg a \mid a \wedge a'$$

$$\beta ::= a \mid \beta \cdot \beta' \mid \beta | \beta' \mid \beta^*$$

Here, α represents a set of actions, more specifically the set $\mathtt{T}$ of all actions, a the set containing a specific action $a \in \mathrm{Act} \cup \{\tau\}$, $\neg a$ its complement and $a \wedge a'$ the set of actions that occur both in a and a'. β represents a set of traces consisting of a set of actions α, the concatenation $\cdot$ of a trace from β and β', the union $|$ of the traces in β and β' and finally β^* the transitive reflexive closure of β, consisting of those traces that a concatenation of finitely many traces from β yields. $\langle\beta\rangle\phi$ is defined to hold if there is a trace β leading to a state s in which ϕ holds. $[\beta]\phi$ is defined to hold if all traces β end up in a state s where ϕ holds.

5. Specification

In this section we present the overall structure of our μCRL specification of TCP, which includes the Data Transfer and Connection Teardown phases as specified in RFCs 793 and 1122, extended to include the Window Scale Option, as specified in RFC 1323. It is the aim of this section to relate the RFCs to the μCRL specification, assisting the reader in bridging the gap between the two.

We do not validate Connection Establishment. As discussed in Section 2, this part of the specification has been well studied in the literature already. While none of these efforts included RFC 1323, the only addition to this phase as proposed by this extension is the communication of variables `SND.WND.SCALE` and `RCV.WND.SCALE` to the other end of the connection. This process is so straightforward that we do not expect it to refute earlier verification efforts of the connection establishment phase.

We instead specify Data Transfer and Connection Teardown, taking the `ESTABLISHED` state as the initial state of our model. We take care to include the core TCP functionality as well as any peripheral functionality that is potentially influenced by the Window Scale Option.

For brevity, we focus most of our attention on the process modelling the TCP instance; this was the primary exercise in modelling TCP. For the complete μCRL specification, including processes modelling the Application and Network Layers, we refer to [35].

Although we specify a generic TCP process that executes the responsibilities of the sending and receiving instances, when composing our model of unidirectional TCP, including processes modelling the Application and Network Layers, some actions must be encapsulated in *TCP1* and *TCP2* to instantiate them as the sending and receiving entities, respectively.

To avoid discussing abstract notions such as connections (at the TCP level) and sessions (at the application level) that are rather detached from their contexts of sending and receiving entities in a network, we will consider a TCP instance that has only one connection with one remote entity. This TCP instance maintains the state of the connection and the TCB. To manage its window, the sender maintains the variables `SND.UNA`, `SND.NXT` and `SND.WND` in the TCB.

5.1. Preliminaries

We adopt the convention of denoting `SND.NXT` as `SND_NXT` in μCRL, and likewise for other variables/states of the RFCs. Furthermore, we use `T` to denote *true* and `F` for *false*. *Nat* reflects the set of natural numbers, on which we define the standard operations $=, +, *, <, \leq, >, \geq$ and mod, the *monus* operation denoted with $\dot{-}$, $\dot{\div}$ denoting integer division and finally a difference operation on sequence numbers modulo n denoted with *seq_diff*.

Of all the data maintained in a segment, only the sequence and acknowledgement number, the window size, the `ACK` and `FIN` flag and the number of octets included in the segment are important to our specification. Hence, a data type *Sgmt* representing segments is defined. On this data type, we define the equality operation $=$ and projections *get_seq_nr*, *get_acknr*, *get_window*, *get_num_octs*, *is_ack* and *fin_flag_set*.

Data is kept in buffers of data type *Buffer* representing a list of natural numbers. Similarly, we define a type *SgmtQueue* for a list of segments. On both data types, we define an operation *first* to get the head; *add* to add an element to the buffer; *add_ordered* to add an element while maintaining the order; *length* to get the number of items on the buffer and *merge* to merge two buffers. In addition, on *Buffer* we define an operation *take_n* which takes one occurrence of an element x from a buffer; *take_set* which takes two buffers b_1 and b_2 and returns a buffer b_3 that consists of the buffer b_1 to which *take_n* is applied for every element in b_2 and *infl* which takes two sequence numbers x and y as arguments, and yields an ordered buffer that contains y sequence numbers starting at x (taking the fact that sequence numbers are taken modulo n into account).

The Transmission Control Block is specified as data type *TransmissionControlBlock*. The variables that it maintains are named according to the RFC and for each of them a getter function is defined. Throughout the specification included in this section, all updates to these variables are denoted as $a \mapsto b$, meaning that the value of variable a is replaced with value b. If a is also used at the right side of the substitution, this means that the old value is first retrieved from the TCB for manipulation.

Finally we define a data type *ConnectionState*, representing all states that a TCP instance may progress through: *ESTABLISHED*, *CLOSE_WAIT*, *FIN_WAIT_1*, *FIN_WAIT_2*, *CLOSING*, *LAST_ACK*, *TIME_ WAIT* and *CLOSED*.

5.2. Data Transfer

We will now present our specification of TCP Data Transfer. The structure of this section is intended to cohere with that of our presentation of the functional specification in Section 3 and Figs. 2 and 3.

5.2.1. Application Layer calls SEND

The first call that is discussed in [1] is the SEND call on pages 56 to 57. By issuing a tcp_rcv_SND call the TCP instance accepts an arbitrary octet and adds it to its send buffer. The call may only be issued if the connection is in state ESTABLISHED or CLOSE_WAIT and if there is send buffer space available for the octet:

$$\begin{aligned}
&TCP(s\colon ConnectionState, t\colon TransmissionControlBlock)\\
&\quad = \sum_{x:Nat} \Big(rcv\_SND(x) \cdot TCP\big(s, t\big[send\_buff \mapsto add\_ordered(x, send\_buff)\big]\big)\\
&\qquad \lhd\ s \in \{ESTABLISHED, CLOSE\_WAIT\} \land x < n \land length(get\_send\_buff(t)) < buff\_capacity\ \rhd\ \delta\Big)
\end{aligned}$$

5.2.2. Octets in send buffer?

In addition, [1] specifies that TCP may segment octets in the send buffer and subsequently send them to the remote entity *"at its own will"*. To this end, we include the following summand in our specification:

$$\begin{aligned}
&+\ call\_SND(sgmt(get\_SND\_NXT(t), get\_RCV\_NXT(t), calc\_wnd(t), can\_send(t), \mathtt{F}, \mathtt{F}))\cdot\\
&\quad TCP\Big(s, t\Big[rtq \mapsto add(sgmt(SND\_NXT, RCV\_NXT, calc\_wnd(t), can\_send(t), \mathtt{F}, \mathtt{F}), rtq),\\
&\qquad\qquad send\_buff \mapsto take\_set(send\_buff, infl(SND\_NXT, can\_send(t))),\\
&\qquad\qquad SND\_NXT \mapsto (SND\_NXT + can\_send(t)) \bmod n\Big]\Big)\\
&\quad \lhd\ s \in \{ESTABLISHED, CLOSE\_WAIT\} \land can\_send(t) > 0\ \rhd\ \delta
\end{aligned}$$

If the connection is in the ESTABLISHED or CLOSE_WAIT state and TCP is allowed to send one or more octets, a segment containing the eligible to be sent octets is passed to the Network Layer by issuing call_SND. This segment is labelled with the sequence number maintained in SND_NXT. After the sequence number, the acknowledgement number and advertised window are included, followed by the number of octets included in the segment and the values of the ACK and FIN flag. The number of octets that TCP can send is the difference m between SND.UNA and SND.NXT. If $m <$ SND.WND, TCP may package $n \leq m$ octets into a segment and send it into the medium. The receive window size relayed in the segment is calculated by applying the scale factor RCV.WND.SCALE to the actual receive window size in the function *calc_wnd*. Subsequently, the octets in the segment are removed from the send buffer, the segment is added to the retransmission queue (*rtq*) and SND_NXT is updated to reflect the next sequence number to be used.

In our model, the ACK flag will always be set to false in segments carrying data octets, and therefore the value of the acknowledgement field in the segment will not be processed by the receiver. The specification dictates that the ACK flag is always set to true and that the latest acknowledgement information is included in each data segment. However, this would complicate the processing of segments in our model and is only pertinent to a bidirectional connection that we do not consider to limit the size of our state space. In a unidirectional setting, the sender's value of RCV_NXT will be constant since it never receives data. Likewise, the receiver's values of SND_NXT and SND_UNA remain constant since it never sends data. Hence, throughout the protocol, if a sender A and a receiver B have agreed on initial sequence number x, then $A_{\mathtt{RCV\_NXT}} = B_{\mathtt{SND\_NXT}} \land B_{\mathtt{SND\_NXT}} = B_{\mathtt{SND\_UNA}} = x$. Acknowledgements will not be processed since $\neg(B_{\mathtt{SND\_UNA}} < A_{\mathtt{RCV\_NXT}})$. If $\mathtt{SND\_UNA} \leq \mathtt{SND\_NXT} < (\mathtt{SND\_UNA} + \mathtt{SND\_WND})$, then the sender is allowed to send $x = (\mathtt{SND\_UNA} + \mathtt{SND\_WND}) - \mathtt{SND\_NXT}$ octets. To this end, we specified a function *can_send* that returns x if the length of the buffer is greater than x, or the length of the buffer otherwise. No octets may be sent if $\neg(\mathtt{SND\_UNA} \leq \mathtt{SND\_NXT} < (\mathtt{SND\_UNA} + \mathtt{SND\_WND}))$. From a modelling perspective, this solves an ambiguity in [1], namely that TCP may send octets *at its own will*.

5.2.3. Segment arrives

All acceptable segments must be processed in the order of the sequence numbers; *rcvr_may_accept* determines a segment acceptable. We distinguish the following cases: the arrival of an unacceptable segment, whether the segment has the FIN flag set, whether it contains acknowledgement information and whether it is carrying data.

The segment is not acceptable RFC 793 explicitly states: *"Segments are processed in sequence. [...] If an incoming segment is not acceptable, an acknowledgment should be sent in reply. [...] After sending the acknowledgment, drop the unacceptable segment and return."* Later, it is suggested that *"Segments with higher beginning sequence numbers may be held for later processing."* However, these segments are held outside of the regular operation of TCP. Hence, implementing this functionality does not add any external behaviour, while having an adverse effect on the size of the state space. Therefore, in our model an unacceptable segment is dropped and an acknowledgement is sent to the sender containing the current value of RCV.NXT. Note that whenever an unacceptable segment is received, an acknowledgement is constructed using the *construct_ack* function. This acknowledgement includes the sequence number of the octet that the TCP instance expects to receive next, the acknowledgement number and the advertised window. The ACK-flag of the acknowledgement segment is set to T while the FIN-flag is set to F.

$$
\begin{aligned}
&+ \sum_{m:Sgmt} \Big(\mathit{call\_RCV}(m) \cdot \mathit{call\_SND}(\mathit{construct\_ack}(t)) \cdot \mathit{TCP}(s,t) \\
&\quad \lhd\ s \in \{\mathit{ESTABLISHED}, \mathit{CLOSE\_WAIT}, \mathit{FIN\_WAIT\_1}, \mathit{FIN\_WAIT\_2}, \mathit{CLOSING}, \mathit{LAST\_ACK}, \mathit{TIME\_WAIT}\} \\
&\qquad \wedge\ (\neg \mathit{rcvr\_may\_accept}(m,t) \vee \mathit{get\_seq\_nr}(m) \neq \mathit{get\_RCV\_NXT}(t))\ \rhd\ \delta \Big)
\end{aligned}
$$

We distinguish between segments carrying data, acknowledgement information and FIN information; we determine the type of a segment using functions *is_ack* and *fin_flag_set*. If neither of these two is true, the segment is understood to be a segment carrying data. Processing is done for each of these situations separately.

The segment contains acknowledgement information If the segment m is an acknowledgement, the TCP instance first checks it is acceptable; *may_accept_ack* verifies whether the acknowledgement number of the segment is strictly between SND_UNA and SND_NXT, or equal to SND_NXT. *must_updt_window* decides whether the window information must be updated, verifying $\mathtt{SND\_WL1} < \mathit{get\_seq\_nr}(m) \vee (\mathtt{SND\_WL1} = \mathit{get\_seq\_nr}(m) \wedge \mathtt{SND\_WL2} \leq \mathit{get\_acknr}(m))$. If so, SND_WND is updated to the size of the window in the segment multiplied by the scale factor SND_WND_SCALE. SND_UNA is updated and segments containing octets with a sequence number of at most i are removed from the retransmission queue, where i is strictly between SND_UNA and the acknowledgement number in the segment, or equal to SND_UNA. In [3] the scale factor is defined as n, resulting in integer division/multiplication by 2^n via bit shifting, whereas we maintain the scale factor as 2^n and apply scaling using division and multiplication. In the first two summands, our call to TCP is parameterised with s' not s: if the TCP instance is in state FIN_WAIT_1, CLOSING or LAST_ACK, the acknowledgement may acknowledge the FIN segment that the TCP instance has sent and the state is updated to FIN_WAIT_2, TIME_WAIT or CLOSED. If the function *must_updt_window* returns *false*, the TCP instance remains in the same state. Finally, if the acknowledgement is not acceptable it is dropped and the TCP instance remains in the same state. An acknowledgement must be returned if $\mathtt{SEG.ACK} > \mathtt{SND.NXT}$. In a unidirectional setting, such a situation will never occur so we exclude such behaviour from our model.

$$
\begin{aligned}
&+ \sum_{m:Sgmt} \Big(\mathit{call\_RCV}(m) \cdot \mathit{TCP}\Big(s', t\Big[\mathit{rtq} \mapsto \mathit{updt\_rtq}(\mathit{rtq}, \mathit{get\_acknr}(m), \mathit{SND\_UNA}), \\
&\qquad\quad \mathit{SND\_WL2} \mapsto \mathit{get\_acknr}(m), \mathit{SND\_WL1} \mapsto \mathit{get\_seq\_nr}(m), \\
&\qquad\quad \mathit{SND\_WND} \mapsto \mathit{get\_window}(m) * \mathit{SND\_WND\_SCALE}, \mathit{SND\_UNA} \mapsto \mathit{get\_acknr}(m) \Big]\Big) \\
&\quad \lhd\ s \in \{\mathit{ESTABLISHED}, \mathit{CLOSE\_WAIT}, \mathit{FIN\_WAIT\_1}, \mathit{FIN\_WAIT\_2}, \mathit{CLOSING}, \mathit{LAST\_ACK}\} \\
&\qquad \wedge\ \mathit{rcvr\_may\_accept}(m,t) \wedge \mathit{get\_seq\_nr}(m) = \mathit{get\_RCV\_NXT}(t) \wedge \mathit{is\_ack}(m) \\
&\qquad \wedge\ \mathit{may\_accept\_ack}(m,t) \wedge \mathit{must\_updt\_window}(m,t)\ \rhd\ \delta \Big)
\end{aligned}
$$

$$
\begin{aligned}
&+ \sum_{m:Sgmt} \Big(\mathit{call\_RCV}(m) \cdot \mathit{TCP}\Big(s', t\Big[\mathit{rtq} \mapsto \mathit{updt\_rtq}(\mathit{rtq}, \mathit{get\_acknr}(m), \mathit{SND\_UNA}), \mathit{SND\_UNA} \mapsto \mathit{get\_acknr}(m) \Big]\Big) \\
&\quad \lhd\ s \in \{\mathit{ESTABLISHED}, \mathit{CLOSE\_WAIT}, \mathit{FIN\_WAIT\_1}, \mathit{FIN\_WAIT\_2}, \mathit{CLOSING}, \mathit{LAST\_ACK}\} \\
&\qquad \wedge\ \mathit{rcvr\_may\_accept}(m,t) \wedge \mathit{get\_seq\_nr}(m) = \mathit{get\_RCV\_NXT}(t) \\
&\qquad \wedge\ \mathit{is\_ack}(m) \wedge \mathit{may\_\ accept\_ack}(m,t) \wedge \neg \mathit{must\_updt\_window}(m,t)\ \rhd\ \delta \Big) \\
&+ \sum_{m:Sgmt} \Big(\mathit{call\_RCV}(m) \cdot \mathit{TCP}(s,t) \\
&\quad \lhd\ s \in \{\mathit{ESTABLISHED}, \mathit{CLOSE\_WAIT}, \mathit{FIN\_WAIT\_1}, \mathit{FIN\_WAIT\_2}, \mathit{CLOSING}, \mathit{LAST\_ACK}\} \\
&\qquad \wedge\ \mathit{rcvr\_may\_accept}(m,t) \wedge \mathit{get\_seq\_nr}(m) = \mathit{get\_RCV\_NXT}(t) \wedge \mathit{is\_ack}(m) \wedge \neg \mathit{may\_accept\_ack}(m,t)\ \rhd\ \delta \Big)
\end{aligned}
$$

The segment is carrying data If the incoming segment is acceptable, and both *is_ack* and *fin_flag_set* return false, it is processed as a data segment. Its octets are added to the receive buffer and RCV_NXT is updated The RCV_ACK_QUEUED flag in the TCB is set to true, indicating an acknowledgement should be sent.

$$
\begin{aligned}
&+ \sum_{m:Sgmt} \Big(\mathit{call\_RCV}(m) \cdot \mathit{TCP}\Big(s, t\Big[\mathit{RCV\_NXT} \mapsto (\mathit{RCV\_NXT} + \mathit{get\_num\_octs}(m)) \bmod n, \\
&\qquad\quad \mathit{rcv\_buf} \mapsto \mathit{merge}(\mathit{rcv\_buf}, \mathit{infl}(\mathit{get\_seq\_nr}(m), \mathit{get\_num\_octs}(m))), \mathit{RCV\_ACK\_QUEUED} \mapsto \mathtt{T} \Big]\Big) \\
&\quad \lhd\ s \in \{\mathit{ESTABLISHED}, \mathit{FIN\_WAIT\_1}, \mathit{FIN\_WAIT\_2}\} \wedge \mathit{rcvr\_may\_accept}(m, \mathit{tcb}) \\
&\qquad \wedge\ \mathit{get\_seq\_nr}(m) = \mathit{get\_RCV\_NXT}(t) \wedge \neg \mathit{is\_ack}(m) \wedge \neg \mathit{fin\_flag\_set}(m)\ \rhd\ \delta \Big)
\end{aligned}
$$

In [1], it is stated that *"when the TCP takes responsibility for delivering the data to the user it must also acknowledge the receipt of the data. [...] This acknowledgement should be piggybacked on a segment being transmitted if possible without incurring undue* *delay"*. [2] clarifies this point further, stating: *"a host [...] can increase efficiency in both the Internet and the hosts by sending fewer than one ACK (acknowledgement) segment per data segment received"*. While the delay that is referred to here is a performance enhancement, its impact is significant enough to justify an increase in the complexity of our model. Therefore, we include this behaviour in our model and do not let the TCP instance send an acknowledgement in the previous summand, but rather let it set a flag that an acknowledgement should be sent. We then include a separate summand that may send an acknowledgement whenever the `RCV_ACK_QUEUED` flag in the transmission control block is set to true. To prevent an acknowledgement from being sent multiple times, this flag is then set to false again. Note that acknowledgement segments are separated from data segments, as we have not specified piggy-backing.

$$+\ call\_SND(construct\_ack(t))\cdot TCP\left(s, t\left[RCV\_ACK\_QUEUED \mapsto \mathtt{F}\right]\right) \ \triangleleft\ get\_RCV\_ACK\_QUEUED(t) = \mathtt{T}\ \triangleright\ \delta$$

This has an additional modelling benefit, solving an ambiguity in [1] and [2] about which window information must be included in the acknowledgement segment. Using separate summands, the size of the receive buffer with the just received segments may be reflected (if the acknowledgement is sent immediately and no `RECEIVE` calls are processed meanwhile) or the size of the receive buffer without the just received segments (if the acknowledgement is only sent after all octets have passed to the Application Layer) and any situation in between.

The segment has the FIN flag set If the incoming segment m is a `FIN` segment, it is processed as described on pages 75–76 of [1]. An acknowledgement is constructed and sent back to the remote end, after which the transmission control block is updated. Note that the state may progress from s to s': if the TCP instance was in state `ESTABLISHED`, `FIN_WAIT_1` or `FIN_WAIT_2` it progresses to state `CLOSE_WAIT`, `CLOSING` or `TIME_WAIT` respectively. In all other cases, the TCP instance stays in the same state. `RCV_ACK_QUEUED` is set to false since we immediately send an acknowledgement a for a `FIN` segment and any outstanding acknowledgements are included in a.

$$\begin{aligned}
+ \sum_{m:Sgmt} \Big(& call\_RCV(m)\cdot call\_SND(construct\_ack(t\left[RCV\_NXT \mapsto (RCV\_NXT+1) \bmod n\right]))\cdot \\
& TCP\left(s', t\left[RCV\_NXT \mapsto (RCV\_NXT+1) \bmod n, RCV\_ACK\_QUEUED \mapsto \mathtt{F}\right]\right) \\
& \triangleleft\ s \in \{ESTABLISHED, FIN\_WAIT\_1, FIN\_WAIT\_2, CLOSE\_WAIT, CLOSING, LAST\_ACK, TIME\_WAIT\} \\
& \quad \land\ rcvr\_may\_accept(m,t) \land get\_seq\_nr(m) = get\_RCV\_NXT(t) \land fin\_flag\_set(m)\ \triangleright\ \delta \Big)
\end{aligned}$$

5.2.4. Retransmission timeout

For each segment the TCP instance puts on the retransmission queue, it starts a timer. When it expires, its segment must be retransmitted. To avoid modelling timing issues in our specification, we abstract away from this timer and allow a TCP instance to retransmit the first element on the retransmission queue at its own convenience at any time. While this behaviour could have a negative impact on the performance of the protocol, for our purposes it does not significantly alter the behaviour compared to a situation in which timers are employed.

$$+\ call\_SND(first(get\_rtq(t)))\cdot TCP(s,t)\ \triangleleft\ length(get\_rtq(t)) > 0\ \triangleright\ \delta$$

5.2.5. Probe zero window

With each acknowledgement, the size of the send window may be adjusted. Adjusting the send window to a size of 0 may lead to a deadlock since as a result of not sending data, the sender will not receive any acknowledgements with an updated window size. Therefore, it must regularly transmit something to the remote entity if `SND_WND` $= 0$.

$$\begin{aligned}
+\ & call\_SND(sgmt(get\_SND\_NXT(t), get\_RCV\_NXT(t), calc\_wnd(t), 1, \mathtt{F}, \mathtt{F}))\cdot \\
& TCP\Big(s, t\Big[rtq \mapsto add(sgmt(SND\_NXT, RCV\_NXT, calc\_wnd(t), 1, \mathtt{F}, \mathtt{F}), rtq), \\
& \qquad\qquad send\_buff \mapsto take\_set(send\_buff, infl(SND\_NXT, 1)), SND\_NXT \mapsto (SND\_NXT+1) \bmod n\Big]\Big) \\
& \triangleleft\ can\_send(t) = 0 \land get\_SND\_WND(t) = 0 \land length(get\_rtq(t)) = 0 \land length(get\_send\_buff(t)) > 0\ \triangleright\ \delta
\end{aligned}$$

If the send window is 0 and the retransmission queue is empty, but octets are available in the send buffer, the sender will construct a segment containing one octet and send it. Again, the octet included in the segment is taken from the send buffer, the segment is put on the retransmission queue and the variable `SND_NXT` is updated. Note that this is the only major difference between our model and the behaviour specified in RFC 793; we delay further explanation and justification of this important revision until Section 6.2.

5.2.6. Application Layer calls RECEIVE

An octet is offered to the Application Layer by issuing a `tcp_rcv_RECEIVE` call, parameterised with the octet pointed at by `RCV_RD_NXT` maintained in the TCB. It is removed from the receive buffer and `RCV_RD_NXT` is incremented. The call may only be issued if the connection is in states `ESTABLISHED`, `FIN_WAIT_1`, `FIN_WAIT_2` or `CLOSE_WAIT`, (`RCV_NXT` − `RCV_RD_NXT`) mod $n > 0$ and the octet with sequence number `RCV_RD_NXT` is available in the receive buffer. In [1], the size of the receive window, stored as `RCV_WND` in the TCB, is updated every time the receive buffer is manipulated. However, page 74 strictly requires the total of `RCV_WND` and `RCV_NXT` not to be reduced. It is unclear whether the total may not be reduced when an incoming segment is processed, or not at all. Either way, we believe that it relates to the requirement that the right edge of the window should never be moved to the left. To simplify the implementation while ensuring this requirement we maintain `RCV_WND` at its initial value, and introduce the variable `RCV_RD_NXT` that is always the sequence number of the next octet to be forwarded to the Application Layer. At all times `RCV_NXT` ≤ `RCV_RD_NXT` ≤ (`RCV_NXT` + `RCV_WND`).

$$\begin{aligned}
+\ & \mathit{rcv\_RECEIVE}(\mathit{get\_RCV\_RD\_NXT}(t))\cdot\\
& \mathit{TCP}\Big(s, t\Big[\mathit{rcv\_buf} \mapsto \mathit{take\_n}(\mathit{rcv\_buf}, \mathit{RCV\_RD\_NXT})\mathit{RCV\_RD\_NXT} \mapsto (\mathit{RCV\_RD\_NXT}+1) \bmod n\Big]\Big)\\
& \lhd\ s \in \{\mathit{ESTABLISHED}, \mathit{FIN\_WAIT\_1}, \mathit{FIN\_WAIT\_2}, \mathit{CLOSE\_WAIT}\}\\
& \quad \wedge\ \mathit{seq\_diff}(\mathit{get\_RCV\_RD\_\ NXT}(t), \mathit{get\_RCV\_NXT}(t)) > 0 \wedge \mathit{length}(\mathit{get\_rcv\_buf}(t)) > 0\\
& \quad \wedge\ \mathit{get\_RCV\_RD\_NXT}(t) \in \mathit{get\_rcv\_buf}(t)\ \rhd\ \delta
\end{aligned}$$

5.3. Connection Teardown

Finally, the TCP instance may receive a `CLOSE` call from the Application Layer, as discussed on pages 60–61 of [1]. The specification states that if such a call is issued by the Application Layer while there are still octets in the send buffer, TCP will queue this call until all of these octets are segmented. Hence, the TCP instance may only perform the `tcp_rcv_CLOSE` action whenever its buffer is empty. The call is processed by sending a segment with the `FIN` flag set, after which the TCP instance will either progress to the `FIN_WAIT_1` or `LAST_ACK` state. The first case models the situation where the connection is still fully opened, while the second case conforms to the situation where the TCP instance has received a `FIN` segment, signalling that the other end has closed the connection.

$$\begin{aligned}
+\ & \mathit{rcv\_CLOSE}\cdot \mathit{call\_SND}(\mathit{sgmt}(\mathit{get\_SND\_NXT}(t), \mathit{get\_RCV\_NXT}(t), \mathit{calc\_wnd}(t), 0, \mathtt{F}, \mathtt{T}))\cdot\\
& \mathit{TCP}\Big(\mathit{FIN\_WAIT\_1}, t\Big[\mathit{rtq} \mapsto \mathit{add}(\mathit{sgmt}(\mathit{SND\_NXT}, \mathit{RCV\_NXT}, \mathit{calc\_wnd}(t), 0, \mathtt{F}, \mathtt{T}), \mathit{rtq}),\\
& \qquad\qquad \mathit{SND\_NXT} \mapsto (\mathit{SND\_NXT}+1) \bmod n\Big]\Big)\\
& \lhd\ s = \mathit{ESTABLISHED} \wedge \mathit{length}(\mathit{get\_send\_buff}(t)) = 0\ \rhd\ \delta
\end{aligned}$$

$$\begin{aligned}
+\ & \mathit{rcv\_CLOSE}\cdot \mathit{call\_SND}(\mathit{sgmt}(\mathit{get\_SND\_NXT}(t), \mathit{get\_RCV\_NXT}(t), \mathit{calc\_wnd}(t), 0, \mathtt{F}, \mathtt{T}))\cdot\\
& \mathit{TCP}\Big(\mathit{LAST\_ACK}, t\Big[\mathit{rtq} \mapsto \mathit{add}(\mathit{sgmt}(\mathit{SND\_NXT}, \mathit{RCV\_NXT}, \mathit{calc\_wnd}(t), 0, \mathtt{F}, \mathtt{T}), \mathit{rtq}),\\
& \qquad\qquad \mathit{SND\_NXT} \mapsto (\mathit{SND\_NXT}+1) \bmod n\Big]\Big)\\
& \lhd\ s = \mathit{CLOSE\_WAIT} \wedge \mathit{length}(\mathit{get\_send\_buff}(t)) = 0\ \rhd\ \delta
\end{aligned}$$

The final event that may occur is the time-wait timeout. A TCP connection may not transfer to the `CLOSED` state – a fictional state that in reality means that the connection no longer exists – before it is absolutely certain that the acknowledgement that it sent in response to a `FIN` segment has been received at the other end. To this end, the connection must be kept alive for at least two times the Maximum Segment Lifetime. Now, if the acknowledgement of the `FIN` segment gets lost, the remote end will eventually retransmit its `FIN` segment. If this segment arrives, the TCP entity will again respond with an acknowledgement and restart the time-wait timer. If eventually this timer goes off, the connection can be closed. As we did with the other timers in the specification, we abstract from this timer as well and include the following summand:

$$+\ \mathit{TW\_TIMEOUT}\cdot \mathit{TCP}(\mathit{CLOSED}, t)\ \lhd\ s = \mathit{TIME\_WAIT}\ \rhd\ \delta$$

stating that if the TCP entity is in the `TIME_WAIT` state, it may progress to the `CLOSED` state. The `TW_ TIMEOUT` action is included since otherwise the recursion would be unguarded, which is not allowed in μCRL. Finally, we must add the following summand since μCRL cannot cope with successfully terminating processes:

$$+\ idle \cdot TCP(s,t) \ \triangleleft \ s = CLOSED \ \triangleright \ \delta$$

stating that if the TCP entity is in the CLOSED state, it may perform the action tcp_idle and recurse. Later, we will ensure that the tcp_idle actions of both entities synchronise. Hence, even once both parties have successfully closed the connection there will be an action to be performed, and as a result of this, the specification does not terminate.

5.4. The complete system

We obtain the complete system by putting two TCP instances in parallel with additional processes modelling the Application and Network Layers. The Application Layer continuously offers octets to the TCP instance by issuing the call al_call_SEND. Receiving data is modelled by having the Application Layer call al_call_RECEIVE for an arbitrary octet. Finally, we specify a Network Layer that may duplicate, reorder and lose data. General action names are renamed into action names specific for each component. We assume the variables to be set as a result of the connection establishment procedure, including the scale factor that each of the TCP instances will apply to their outgoing segments. When instantiating processes for unidirectional TCP, we encapsulated (i.e., blocked) both AL2_call_SEND and TCP2_rcv_SEND to prevent AL2 from issuing the SEND call. Similarly, AL1_call_RECEIVE and TCP1_rcv_RECEIVE were encapsulated to prevent AL1 from issuing the RECEIVE call, which would yield a non-terminating model of the Data Transfer phase of TCP including the Connection Teardown procedure.

For reasons discussed in Section 3.3.1, we need not incorporate sequence number reuse since TCP cycles through all of its sequence numbers, and waits until all segments have been acknowledged and all duplicates have drained from the network, before starting a new run with a previously used sequence number. Furthermore, to avoid the need for timing restrictions in our specification, we limit our verification to one run of sequence numbers. We may still start anywhere in the sequence number space, since all calculations are defined modulo the size of this space. However, the following problem remains. Assume a sequence number space with range $m \dots n$. The receiving TCP instance will still accept an octet with sequence number m after receiving the octet with sequence number n. Hence, if such an octet is still in the medium as a result of duplication or retransmission, it will be accepted by the receiving TCP instance upon receipt and subsequently delivered to the Application Layer. Given that the assumption on the Maximum Segment Lifetime holds, such behaviour cannot occur in a real-world situation. To model this we ensure that the global variable maintaining the total number of sequence numbers is greater than the number of octets, which guarantees that the problem will not occur.

Our verification concerns the correctness of TCP and not its performance. Therefore, our μCRL specification does not include the following performance enhancing features from RFC 1122: algorithms to avoid the Silly Window Syndrome as discussed on pages 89 and 97–100; improvements to the calculation of the retransmission timer (page 90); support for repackaging the segments on the retransmission queue (page 91); the half-duplex close sequence; nor the reopening of a connection during the close sequence (page 88). We did include the corrections to the TCP connection state diagram related to the Connection Teardown procedure (page 86), the probing of zero windows (page 92), the acceptance criteria for incoming acknowledgements (page 94) and the remarks on when to send an acknowledgement segment (page 96). Of the features in RFC 793 involving Data Transfer, the urgent data function is omitted because its goal is unclear. Either it is used to relay out-of-band data, or to stimulate the Application Layer at the receiving end to issue the RECEIVE call. Either way, it does not alter the behaviour of (regular) data processing at the receiving end. Furthermore, the push function is excluded because the de facto standard programming interface to TCP, the sockets API, does not include support for this function [36].

Next we discuss our verification of the Data Transfer and Connection Teardown phases. We found the state space of our specification of the two phases combined to be too large to generate to perform a single verification.

6. Verification of the Data Transfer phase

Our verification of the Data Transfer phase focused on two aspects of our model: (i) we verified that its state space is deadlock-free, and (ii) we compared the external behaviour of our model, defined in terms of the SEND and RECEIVE calls issued by the Application Layer, to the external behaviour of a FIFO queue. One can consider the SEND call of TCP as putting something into a queue and RECEIVE as taking something from it: the sender puts data elements into the transport medium and the receiver takes them all out of this medium in precisely the same order. We first obtained a model $TCP_{\rightarrow}$ from the system specified in Section 5.4 and excluded connection termination by encapsulating actions AL1_call_CLOSE, AL2_call_CLOSE, TCP1_rcv_CLOSE and TCP2_rcv_CLOSE, to ensure that they will not be called in our model. We then specified a behavioural specification B, and generated an LTS from both B and $TCP_{\rightarrow}$. All actions in $TCP_{\rightarrow}$, other than SEND and RECEIVE, were defined as internal behaviour. We minimised the LTS of $TCP_{\rightarrow}$ and verified that it is branching bisimilar to the LTS of B: $TCP_{\rightarrow} \leftrightarrow_B B$. Note that a fairness assumption is enforced by the minimisation algorithm; τ-loops from which an ‘exit’ is possible are eliminated from our minimised state space. Such τ-loops arise from segments that are continuously dropped by the Network Layer or a sequence of repeated retransmissions, behaviour that we can safely abstract from.

For both $TCP_{\rightarrow}$ and B, we generated a state space using the distributed state space generation tool lps2lts-dist of the LTSmin toolset [6]. By using the --deadlock option, absence of deadlocks could be checked during state space generation. In addition, we used the --cache option to speed up state space generation. State space generation was run

Table 2
Statistics of the state space generation for our model.

Octets sent	Window size	Window scale	Medium capacity	Levels	States	Transitions	Exploration time
4	2	1	2	36	881.043	3.910.863	21 sec
			3	40	11.490.716	53.137.488	104 sec
			4	44	91.821.900	434.372.541	7.5 min
8	2	1	2	54	16.126.380	76.356.475	3 min
			3	58	823.501.590	4.031.264.559	49 min
	4	2	2	49	98.697.902	473.332.511	15 min
			3	56	3.505.654.685	Counter Overflow	3 hrs, 40 min
16	4	2	2	77	3.255.174.492	Counter Overflow	4 hrs, 40 min

on the DAS-4 cluster, more specifically on eight nodes equipped with an Intel Sandy Bridge E5-2620 processor clocked at 2.0 GHz, 64 gigabytes of memory and a K20m Kepler GPU with 6 gigabytes of on-board memory. At each node, we utilised only one core to prevent the process from running out of memory. Table 2 shows some benchmarks of the state space generation for $TCP_{\rightarrow}$ for several different parameterisations. Subsequently, the state space of $TCP_{\rightarrow}$ was minimised with the `lts-reduce-dist` tool of the `LTSmin` toolset, which uses the distributed minimisation algorithm as described in [37]. Finally, the `ltsmin-compare` tool was used to verify that $TCP_{\rightarrow} \leftrightarrows_B B$.

We performed our verification assuming a sequence number space of size $2^3 + 1$, a window size of 2^2, a scale factor of 2^1 and a medium capacity of 2^1 segments, in which the sending TCP sends 2^3 octets. With these parameters, we obtained a model that was small enough to verify within reasonable time, with characteristics that are representative for a real-world implementation of TCP. If the size of the model increases, all relevant buffers and calculations will simply scale with this increase; it is unlikely that errors are introduced as a result. The capacity of the medium significantly impacts state space size. Since one segment may contain at most as many segments as the size of the window, a medium capacity of 2 means that a TCP instance can send at most two windows of data segments into the medium before it must 'wait' for the medium to deliver or lose a segment. For window scaling to be non-trivial, the window size should be at least 2^2 with a scale factor of 2^1, allowing three possible sizes, zero, two and four that allow for interesting scenarios where the reported size of the window is shrunk to half its original size.

6.1. Correctness of the Window Scale Option

Our initial hypothesis was that as windows sizes are reported in units of 2^n when implementing the Window Scale Option, problems could arise when a single octet is sent and the window reported by the receiving TCP entity must be adapted. Conceivably, a sending entity could have a view of the size of the window at the receiving end that exceeds the maximum buffer space available. With the aid of our formal specification, we find that this is not the case. Both entities maintain the send and receive window as 32-bit numbers and maintain a scale factor by which they right/left-shift the value reported in/taken from an acknowledgement segment. This shift by a factor k has the same effect as a floored division or multiplication by a factor 2^k. Assume a receive buffer capacity 2^{n+1} and, therefore, a window size of at most 2^n and a scale factor k, where $0 < k \leq (n-1)$, resulting in a division or multiplication by 2^k. If the receiver receives a segment carrying $0 < m \leq 2^n$ bytes, two scenarios may occur: (i) the reduced buffer space (receive window) is reported in the acknowledgement; or, (ii) the old buffer space is reported. If (i), then the reported window size is $\lfloor(2^n - m)/2^k\rfloor < 2^{n-k} < 2^n$ else if (ii), then nothing changes, the reported window size is $\lfloor 2^n/2^k \rfloor \leq 2^{n-k} < 2^n$. The reported buffer size is always $\leq 2^{n-k}$ and can never become greater than $2^{(n-k)+k} = 2^n$ when it is left-shifted at the remote end. A sending entity never views the receive buffer space available at a receiving entity to exceed the maximum buffer space available.

A second conceivable problem relates to [1] stating that a TCP instance should not 'shrink' its receive window reducing the buffer capacity, i.e., the right edge of the window is moved to the left. Assume a sender and receiver have agreed upon a window size of 4. The sender sends two octets and then immediately sends another two octets. By the arrival of the first segment at the receiver, the octets are put in its receive buffer and, unfortunately, at the same time the capacity of the buffer is also reduced by one octet, causing the receiving entity to report a window of size 1 to the sender rather than 2. As the second segment, carrying two octets, is already in transit, it will be discarded upon arrival at the receiver, because it contains more octets than the receiver may accept. The sender will keep retransmitting this segment and it will be discarded as long as no octets are taken from the receive buffer. If window sizes get bigger, the delay incurred may significantly impact the performance of the protocol. Eventually, however, the octet will be accepted when buffer space becomes available as octets that arrived earlier are taken from the receive buffer.

When window scaling is in effect, one might expect such a scenario to occur every time an odd number of bytes is sent, due to the size of the window being reported only in multiples of 2^n. However, in this case the actual capacity of the receive buffer is not reduced and the receiver maintains the window size as a 32-bit and not a 16-bit number. Therefore, the second segment, which may have been in transit already, will still be accepted and an acknowledgement containing the latest size of the window will be sent back within reasonable time. Where the segment was not yet sent, the difference in the number of octets that may be sent is only 1, causing a performance not a correctness issue.

6.2. Recommended revision of RFC 793

During our verification, we ran into a deadlock resulting from a misinterpretation of the RFC 793 specification, as we will show below. Therefore, we recommend revising the specification in its dealing with zero windows; requiring that *whenever the sender (i) has data on its send buffer, (ii) has a zero window and (iii) has an empty retransmission queue, a segment is sent to probe the zero window containing at least one octet of data from the send buffer.* This behaviour was included in our model as stated in Section 5, but we withheld explanation and justification until now.

Instead, the current specification [1] states on page 42 that: *"The sending TCP must be prepared to accept from the user and send at least one octet of new data even if the send window is zero. The sending TCP must regularly retransmit to the receiving TCP even when the window is zero. [...] This retransmission is essential to guarantee that when either TCP has a zero window the re-opening of the window will be reliably reported to the other."* The latter part of this statement is confusing. It is not the *retransmission* that is essential, but rather the *transmission* of a segment (whether taken from the send buffer or the retransmission queue) when the send window is zero.

To see why, suppose that the sender has two octets on its send buffer and sends only the first of these. The receiver then acknowledges this octet, but does not yet take it from its buffer. As a result of this, both the send and receive window are now 0. In this scenario, there is still data to be sent, but the retransmission queue is empty. If the requirements above are strictly followed, the zero window will never be probed as long as the user does not provide *new data* from which the sender can accept and send at least one octet, and therefore leads to deadlock. Implicitly, the reader may expect data on the send buffer to be sent in this case, regardless. However, this is certainly contradicted by the suggestion to *"avoid sending small segments by waiting until the window is large enough before sending data"*. Note that care should be taken when implementing this feature, since as a result of waiting to send something, no new acknowledgements will arrive to update the window information, again leading to deadlock.

Requiring the sender to be prepared to send at least one octet of new data even when the retransmission queue is non-empty also ensures that the window will be reopened, but not in the way one would expect. The new data is sent to the receiver, which rejects the out of sequence segment. However, as a result, the receiver sends an acknowledgement containing up-to-date window information, potentially reopening the window. Our proposed revision intentionally does not attend to the case of the retransmission queue being non-empty; it is already covered by the requirement that *"if the retransmission timeout expires on a segment in the retransmission queue, send the segment at the front of the retransmission queue again, reinitialize the retransmission time and return"* on page 77 of [1]. As an advantage, whenever the retransmission queue is non-empty, an in-sequence segment will be sent and therefore accepted while its acknowledgement may also reopen the window. Only if the retransmission queue is empty, a segment containing new data probes the zero window. This segment is guaranteed to be accepted if the receiver has reopened its window.

It may be that our revision matches the interpretation intended of the original specification, but we have experienced that its wording can lead to implementations that deadlock. A formal specification, given here in μCRL, clearly leaves less scope for erroneous implementations due to misinterpretation.

7. Verification of Connection Teardown

In the previous section, we verified Data Transfer in isolation of Connection Teardown to avoid untenable state space explosion. To keep some notion of data transfer in our verification of Connection Teardown, we opted for a verification where one of the two TCP entities is required to send one octet of data before it could close its connection. Combined with our earlier verification, which showed that all octets that are buffered at the sender are eventually received, this scenario indicates that a connection will not be closed before all data is delivered.

Again, we had to prevent deadlock scenarios and undesired behaviour as a result of the reuse of sequence numbers from occurring in the state space as generated from our model. To understand that this problem arises again, recall from our discussion in Section 3 that the FIN segments also consume sequence numbers. The solution is now also easy to understand: rather than using $n+1$ sequence numbers to send n octets, we uses $n+3$ sequence numbers to account for the sequence numbers used for the FIN segments.

We obtained a model from our *SystemSpecification* in a similar fashion as in the previous section, although we now also encapsulate the actions AL1_call_CLOSE, TCP1_rcv_CLOSE, AL2_call_CLOSE and TCP2_rcv_CLOSE to ensure that they only occur in synchrony. However, our original specification in Section 5.3 contains a flaw. When both *TCP1* and *TCP2* are in the CLOSING state, they have sent and received a FIN segment and sent the acknowledgements thereof. When the acknowledgement sent by *TCP1* arrives at *TCP2*, it progresses to TIME_WAIT and immediately to the CLOSED state as we have not explicitly modelled timeouts. If the acknowledgement sent by *TCP2* gets lost, *TCP1* is forever stuck in the CLOSING state, retransmitting its FIN segment. On page 22 of [1] it is stated that the TIME_WAIT state *"represents waiting for enough time to pass to be sure the remote TCP received the acknowledgement of its connection termination request"*. The connection must be kept 'half-open' as long as the remote entity may try to retransmit its FIN segment, to ensure that an acknowledgement thereof will eventually arrive. To fix this flaw in our model without adding timing aspects, we ensured that actions tcp_TW_TIMEOUT of *TCP1* and *TCP2* occurred in synchrony i.e., *TCP1* and *TCP2* arrive in the TIME_WAIT state before the connection can progress to the CLOSED state. However, as connections can progress through the LAST_ACK state instead of the TIME_WAIT state during the closing procedure, we introduced an additional state LAST_ACK2 and adapted our specification such that the TCP instance will progress from LAST_ACK to LAST_ACK2 rather than CLOSED. Finally, we adapted the penultimate summand of Section 5.3 as follows:

$$+\ \mathtt{TW\_TIMEOUT} \cdot \mathit{TCP}(\mathit{CLOSED}, t) \ \triangleleft\ \ s = \mathit{TIME\_WAIT} \vee s = \mathit{LAST\_ACK2}\ \ \triangleright\ \delta$$

It is important to state that by the fact that our model does not include connection establishment, the Connection Teardown procedure is only verified starting from the ESTABLISHED state. Scenarios where a connection is closed during connection establishment, before both ends have reached the ESTABLISHED state, are not included. As discussed in Section 3, connections are closed in a simplex fashion. If one of the entities closes its connection, indicating that it has no more data to send, it must still accept segments from the remote end, and not progress to the CLOSED state until the other end has also indicated it has no more data to send.

Recall that (i) the TCP instance only accepts a CLOSE call from the Application Layer if it has no more octets to send, (ii) the FIN segment has a sequence number $\geq i + 1$ if i was the sequence number of the last data octet that was sent over the connection, and (iii) a receiving TCP instance only accepts segments with sequence number $i = \mathtt{RCV.NXT}$. Since the CLOSED state can only be reached after accepting a FIN segment (see Fig. 1), we can conclude that a receiving TCP instance will never reach the CLOSED state without having accepted all data octets that were buffered at the sender at the time that the CLOSE call was issued. However, this does not yet guarantee that connections will be closed whenever an Application Layer issues the CLOSE call to its TCP instance. To verify this, we had to check several properties on the state space generated from our model of TCP with Connection Teardown.

The first of these properties, formulated as a regular μ-calculus formula, states that whenever *TCP1* accepts the CLOSE call from the Application Layer, our model will eventually end up in a state from which it may perform the CONNECTION_CLOSED transition. From our discussion of our specification, we know that this transition is only enabled if both TCP instances are in the CLOSED state.

$$[\mathtt{T}^* \cdot \mathtt{TCP1\_CLOSE}]\mu X \cdot (\langle \mathtt{T} \rangle \mathtt{T} \wedge [\neg \mathtt{CONNECTION\_CLOSED}]X) \tag{1}$$

We verified the same for *TCP2*. Taken together, these properties intuitively state *"whenever either of the TCP instances accepts the* CLOSE *call from the Application Layer, the connection will eventually end up in the* CLOSED *state"*. Hence, it is a *liveness* property. In addition, we verified that both entities must accept the CLOSE call from their Application Layer before the connection may be closed. To this end, we checked the following *safety* property:

$$[(\neg \mathtt{TCP1\_CLOSE})^* \cdot \mathtt{CONNECTION\_CLOSED}]\mathtt{F} \tag{2}$$

The property states that the connection can never reach a state in which it can perform a CONNECTION_CLOSED transition if *TCP1* does not accept the CLOSE call from its Application Layer. Again, we also verified this for *TCP2*. Taken together, these properties ensure that both entities must accept the CLOSE call from their Application Layer before the connection ends up in the CLOSED state. Finally, we verified that our state space does not contain deadlocks as these again signal a problem for the same reasons as discussed before.

We used the lpo2lps-dist tool of the LTSmin toolset to generate a state space from the linear process equation obtained from our μCRL specification, in a distributed fashion. State space generation was again performed on 8 DAS-4 nodes equipped with an Intel Sandy Bridge E5-2620 processor clocked at 2.0 GHz, 64 gigabytes of memory and a K20m Kepler GPU with 6 gigabytes of on-board memory. It took around two minutes to generate a state space consisting of 42 levels, 3.296.792 states and 11.010.169 transitions. During state space generation, it was verified that there are no deadlocks. After the state space generation, we again minimised the state space modulo branching bisimilarity, using the ltsmin-reduce-dist tool, and finally, we checked the aforementioned properties using the CADP toolset. All four properties, as well as absence of deadlocks, were proven to hold for our model.

8. Conclusion

TCP plays an important role in the Internet, providing reliable transport of data over possibly faulty networks. The protocol is complex and its specification consists of many documents that mainly describe the proposed functioning of the protocol in natural language. We set out to formally specify TCP extended with the Window Scale Option and verify its correctness, redressing the lack of consideration paid to this option in earlier verification efforts.

We have recommended revisions to RFC 793 to prevent any misinterpretation of how and when to probe the zero window, which we have shown can lead to deadlock. Such misinterpretation is inherent to specifications expressed in natural language. A formal specification provides a more precise reference of the intended function of a protocol. Our μCRL specification may serve as a useful reference for implementors; we have taken care to bridge the gap between the functional specification of TCP and our μCRL specification, providing sufficient documentation to reproduce our results. We found μCRL to be sufficiently powerful to express all pertinent components of TCP, including the Window Scale Option of primary concern in this paper and believe that the unidirectional instances that we have verified are general enough to carry over to larger parameterisations without the introduction of errors. However, we do acknowledge the fact that using control information in the opposite direction is a source of potential errors and requires verification. We therefore set out to specify bidirectional data transfer but had to abandon a verification as a result of intractable state space explosion. We also split our verification efforts into distinct verifications of the Data Transfer and Connection Teardown phases of TCP, so that generating the state space from our more general specification remained feasible.

Using the `LTSmin` toolset, we were able to formally verify that our μCRL specification of unidirectional TCP extended with the Window Scale Option does not contain deadlocks, and that its external behaviour is branching bisimilar to a FIFO queue for a significantly large instance. Using the `CADP` toolset, we also showed that if two TCP entities are in the `ESTABLISHED` state and either of them accepts the `CLOSE` call from the Application Layer, the connection will eventually be closed. Additionally, we showed that the connection can only be closed if both entities accept the `CLOSE` call from the Application Layer.

Although we could only verify the TCP for relatively small parameter values, we believe that the specification is general enough to make the introduction of errors as parameters are increased highly unlikely. In particular, with regard to sequence numbers two issues could introduce errors: reuse and wrapping. Firstly, sequence number reuse is not really critical for the window scale option if the conditions imposed on RFC 1323 are followed strictly, as discussed in Section 3.3.1. Secondly, and more importantly, the largest sequence number size we managed to model check covers every possible scenario of sequence number wrapping, where the sequence number transposes from $n-1$ to 0 in case of a sequence number space of size n.

Still, it would clearly be highly desirable to be able to model check the protocol for larger parameter values, and include features like bidirectional Data Transfer and Connection Teardown into one specification. However, the state explosion problem inevitably means that even significant advances in explicit state model checking will have a limited impact in this respect. Techniques like symbolic model checking, abstraction and partial order reduction are essential to push model checking toward ever larger applications.

Acknowledgements

The authors are indebted to Dr. Barry M. Cook, for posing the initial research question, and Dr. Kees Verstoep, for essential support in using the DAS-4 cluster.

References

[1] J. Postel, Transmission control protocol, RFC 793.
[2] R. Braden, Requirements for Internet hosts-communication layers, RFC 1122.
[3] V. Jacobson, R. Braden, D. Borman, TCP extensions for high performance, RFC 1323.
[4] B. Badban, W. Fokkink, J. Groote, J. Pang, J. van de Pol, Verification of a sliding window protocol in μCRL and PVS, Form. Asp. Comput. 17 (3) (2005) 342–388.
[5] B. Badban, W. Fokkink, J. van de Pol, Mechanical verification of a two-way sliding window protocol, in: CPA, in: CSE, vol. 66, IOS Press, 2008, pp. 179–202.
[6] S. Blom, J. van de Pol, M. Weber, LTSmin: distributed and symbolic reachability, in: CAV, in: LNCS, vol. 6174, Springer, 2010, pp. 345–359.
[7] L. Lockefeer, D.M. Williams, W.J. Fokkink, Formal specification and verification of TCP extended with the window scale option, in: FMICS, in: LNCS, vol. 8718, Springer, 2014, pp. 63–77.
[8] S. Murphy, A. Shankar, Service specification and protocol construction for the transport layer, in: SIGCOMM, ACM, 1988, pp. 88–97.
[9] M. Smith, Formal verification of communication protocols, in: FORTE, in: IFIP Conf. Proc., vol. 69, Chapman & Hall, 1996, pp. 129–144.
[10] M. Smith, Formal verification of TCP and T/TCP, Ph.D. thesis, Massachusetts Institute of Technology, 1997.
[11] M. Smith, K. Ramakrishnan, Formal specification and verification of safety and performance of TCP selective acknowledgement, IEEE/ACM Trans. Netw. 10 (2) (2002) 193–207.
[12] I. Schieferdecker, Abruptly terminated connections in TCP – a verification example, in: Applied Formal Methods in System Design, 1996, pp. 136–145.
[13] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2011: a toolbox for the construction and analysis of distributed processes, Int. J. Softw. Tools Technol. Transf. 15 (2) (2013) 89–107.
[14] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, K. Wansbrough, Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets, in: SIGCOMM, ACM, 2005, pp. 265–276.
[15] J. Billington, B. Han, On defining the service provided by TCP, in: ACSC, in: CRPIT, vol. 16, ACS, 2003, pp. 129–138.
[16] B. Han, J. Billington, Validating TCP connection management, in: SEFW, ACS, 2002, pp. 47–55.
[17] B. Han, J. Billington, Experience using coloured Petri nets to model TCP's connection management procedures, in: CPN, 2004, pp. 57–76.
[18] B. Han, J. Billington, Termination properties of TCP's connection management procedures, in: ICATPN, in: LNCS, vol. 3536, Springer, 2005, pp. 228–249.
[19] T. Ridge, M. Norrish, P. Sewell, A rigorous approach to networking: TCP, from implementation to protocol to service, in: FM, in: LNCS, vol. 5014, Springer, 2008, pp. 294–309.
[20] M. Bezem, J. Groote, A correctness proof of a one-bit sliding window protocol in μCRL, Comput. J. 37 (4) (1994) 289–307.
[21] E. Madelaine, D. Vergamini, Specification and verification of a sliding window protocol in LOTOS, in: FORTE, in: IFIP Trans., vol. C-2, 1991, pp. 495–510.
[22] D. Chkliaev, J. Hooman, E. de Vink, Verification and improvement of the sliding window protocol, in: TACAS, in: LNCS, vol. 2619, Springer, 2003, pp. 113–127.
[23] A. Tanenbaum, Computer Networks, 4th ed., Prentice Hall, 2002.
[24] S. Floyd, J. Mahdavi, M. Mathis, A. Romanow, TCP selective acknowledgment options, RFC 2018.
[25] W. Fokkink, Modelling Distributed Systems, Texts in Theoretical Computer Science, An EATCS Series, Springer, 2007.
[26] D. Peled, Software Reliability Methods, Springer, 2001.
[27] R. van Glabbeek, The linear time – branching time spectrum I – the semantics of concrete, sequential processes, in: CONCUR, in: LNCS, vol. 458, Springer, 1990, pp. 278–297.
[28] R. van Glabbeek, The linear time – branching time spectrum II, in: CONCUR, in: LNCS, vol. 715, Springer, 1993, pp. 66–81.
[29] R. Milner, A Calculus of Communicating Systems, LNCS, vol. 92, Springer, 1980.
[30] R. van Glabbeek, W. Weijland, Branching time and abstraction in bisimulation semantics, J. ACM 43 (3) (1996) 555–600.
[31] M. Kwiatkowska, Survey of fairness notions, Inf. Softw. Technol. 31 (7) (1989) 371–386.

[32] J. Groote, F. Vaandrager, An efficient algorithm for branching bisimulation and stuttering equivalence, in: ICALP, in: LNCS, vol. 443, Springer, 1990, pp. 626–638.
[33] D. Kozen, Results on the propositional μ-calculus, in: ICALP, in: LNCS, vol. 140, Springer, 1982, pp. 348–359.
[34] R. Mateescu, M. Sighireanu, Efficient on-the-fly model-checking for regular alternation-free mu-calculus, Sci. Comput. Program. 46 (3) (2003) 255–281.
[35] L. Lockefeer, Formal specification and verification of TCP extended with the window scale option, Master's thesis, VU University Amsterdam, 2013, http://www.cs.vu.nl/~wanf/theses/lockefeer.pdf.
[36] W. Stevens, TCP/IP Illustrated, vol. 1: The Protocols, Addison-Wesley, 1994.
[37] S. Blom, S. Orzan, Distributed state space minimization, Int. J. Softw. Tools Technol. Transf. 7 (3) (2005) 280–291.